Combinaison des techniques de Bounded Model Checking et de programmation par contraintes pour l'aide à la localisation d'erreurs : exploration des capacités des CSP pour la localisation d'erreurs

Mohammed Bekkouche

UNIVERSITÉ NICE SOPHIA ANTIPOLIS

ÉCOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H È S E

pour obtenir le titre de

Docteur en Sciences

de l'Université Nice Sophia Antipolis
Mention : INFORMATIQUE

Présentée et soutenue par

Mohammed BEKKOUCHE

Combinaison des techniques de Bounded Model Checking et de Programmation par Contraintes pour l'aide à la localisation d'erreurs

Thèse dirigée par Michel RUEHER et Hélène COLLAVIZZA

préparée à l'I3S (UNSA/CNRS)

soutenue le 11 décembre 2015

Jury :

Rapporteurs	**Laurence PIERRE**	Professeur de l'Université Joseph Fourier, Grenoble, France
	Fabrice BOUQUET	Professeur d'informatique, Université de Franche-Comté, France
Directeur	**Michel RUEHER**	Professeur à l'université de Nice Sophia Antipolis, France
Co-Directeur	**Hélène COLLAVIZZA**	Maître de Conférences de Polytech'Nice Sophia-Antipolis, France
Président	**Frédéric PRECIOSO**	Professeur à l'université de Nice Sophia Antipolis, France
Invités	**Ulrich JUNKER**	Scientifique à la résolution avancée des problèmes, ILOG, France
	Yahia LEBBAH	Professeur à l'université d'Oran Es-Sénia, Algérie

A mon père,
mes frères et soeur

Remerciements

J'aimerais tout d'abord remercier infiniment mon directeur de thèse, M. Michel RUEHER, pour m'avoir fait confiance, guidé, encouragé et conseillé tout au long de ce travail de recherche doctoral. Je souhaiterais aussi exprimer ma gratitude à ma codirectrice de thèse, Mme. Hélène COLLAVIZZA, pour sa gentillesse, tous ses conseils et toute son aide au cours de l'élaboration de cette thése. Ils m'ont beaucoup appris grâce à leur rigueur et compétence scientifique.

Mes remerciements vont également à M. Michel RIVEILL, directeur du laboratoire d'Informatique, Signaux et Systèmes de Sophia Antipolis (I3S/CNRS), ainsi que M. Gilles BERNOT, directeur de l'École Doctorale Sciences et Technologies de l'Information et de la Communication (ED STIC), pour m'avoir accueilli au sein de ces institutions. Je remercie également Mme. Régine SAELENS, secrétaire de l'ED STIC.

Je voudrais exprimer mes remerciements les plus sincères aux membres de mon jury, qui ont accepté d'évaluer ce travail de thèse.

Je remercie grandement le Ministére de l'Enseignement Supérieur et de la Recherche (MESR) qui a financé cette thèse en m'accordant un poste d'allocataire de recherche.

Je tiens à remercier sincèrement MM. Hiroshi HOSOBE, Yahia LEBBAH, Bertrand NEVEU, You LI, Manu JOSE, Si-Mohamed LAMRAOUI et Shin NAKAJIMA pour les échanges fructueux que nous avons eus. Je tiens aussi à remercier M. Olivier PONSINI pour son aide et ses précieux conseils lors de la réalisation du prototype de notre systéme. Je remercie également Clément Poncelet pour ses corrections utiles sur ce manuscrit.

Je remercie beaucoup M. Frédéric PRECIOSO d'avoir accepté de me rencontrer et de discuter avec moi de ma thèse.

J'adresse aussi mes remerciements à tous les membres de l'équipe projet MDSC/CeV (Contraintes et Vérification) du laboratoire I3S, MM. Claude MICHEL, Mohammed Said BELAID, Emmanuel KOUNALIS, Sylvain LIPPI et Mme. Carine FÉDÈLE. Je souhaite remercier spécialement et vivement Mme. Sandra DEVAUCHELLE, notre assistante préférée. Je remercie aussi M. Jean Charles Régin, M. Arnaud MALAPERT, Guillaume PEREZ et Mohammed REZGUI.

Je voudrais remercier tous mes collégues de l'équipe MDSC/I3S et de laboratoire. Je pense à l'ensemble des thésards et chercheurs. Je remercie à cette occasion mes collègues de bureau Benjamin MIRAGLIO et Emilien CORNILLON. Merci à Jonathan BEHAEGEL, Lam HUNG, Fernando IRETA, Pierre-Alain SCRIBOT, Simon MARTIEL, Alexis ZUBIOLO, Youssef BENNANI, Youssef ALJ.

Enfin, je remercie chaleureusement mon père, mes frères et soeur, Abdelouahab, Yahya, Zakaria et Meriem, pour leur soutien moral au cours de ces trois années de thèse. Je remercie également tout le reste de ma famille : mes oncles, tantes, cousins et cousines.

Table des matières

Chapitre 1

Introduction

*"L'humain est incomplet, il est incapable d'examiner les conséquences de ce qu'il fait. L'ordinateur, au contraire, exécute toutes les conséquences de ce qui est écrit. Si jamais dans la chaîne de conséquences, il y a quelque chose qui ne devrait pas être là (erreur humaine), l'ordinateur va exécuter le programme. Voilà le **bug**. Un être humain est incapable de tirer les conséquences de ses actions à l'échelle de milliards d'instructions. Par contre, ceci est ce que la machine va faire, elle va exécuter des milliards d'instructions."* – Gérard Berry

Sommaire

1.1 Motivations de la Thèse

Nous ne pourrons jamais nous débarrasser des vulnérabilités. Les vulnérabilités sont essentiellement des bugs [1], et nous aurons toujours des bugs, car les programmes que nous utilisons sont écrits par des êtres humains. Or les êtres humains feront toujours des erreurs, donc, les erreurs sont inévitables dans un programme [2]. Elles peuvent nuire à son bon fonctionnement et avoir des conséquences financières extrêmement graves. Ainsi, elles constituent une menace pour le bien-être humain. À propos des conséquences de la présence des erreurs dans un programme, Gérard Berry [3] dit : "L'humain est incomplet, il est incapable d'examiner les conséquences

1. Les termes erreur, bug et faute sont souvent utilisés indifféremment.
2. Je cite dans le lien suivant des histoires récentes de bugs logiciels : http://www.i3s.unice.fr/~bekkouch/Bug_stories.html
3. http ://www.college-de-france.fr/site/gerard-berry/

de ce qu'il fait. L'ordinateur, au contraire, exécute toutes les conséquences de ce qui est écrit. Si jamais dans la chaîne de conséquences, il y a quelque chose qui ne devrait pas être là (erreur humaine), l'ordinateur va exécuter le programme. Voilà le **bug**. Un être humain est incapable de tirer les conséquences de ses actions à l'échelle de milliards d'instructions. Par contre, ceci est ce que la machine va faire, elle va exécuter des milliards d'instructions." Conséquemment, le processus de débogage (la détection, la localisation et la correction d'erreurs) est essentiel. Pour le débogage, les programmeurs doivent d'abord être capables d'identifier exactement l'emplacement des erreurs (localisation d'erreurs) ; puis trouver un moyen de les fixer (correction d'erreurs) [Wong 2010]. La motivation principale de la localisation d'erreurs est de diminuer le coût de la correction des bugs.

Dans le développement de logiciels, le débogage peut être l'activité la plus difficile. De plus, la localisation d'erreurs est l'étape qui coûte le plus sur les trois aspects de débogage : la localisation d'erreurs représente environ 95% de l'activité de débogage [Paul 2013]. En effet, quand un programme P est non conforme vis-à-vis de sa spécification (P contient des erreurs), un vérificateur de modèle peut produire une trace d'un contre-exemple, qui est souvent longue et difficile à comprendre même pour les programmeurs expérimentés. Cette thèse propose une nouvelle approche basée sur la programmation par contraintes pour l'aide à la localisation d'erreurs dans un programme pour lequel un contre-exemple a été trouvé ; c'est-à-dire pour lequel on dispose d'une instanciation des variables d'entrée qui viole la postcondition.

1.2 Problématique de la localisation d'erreurs

La localisation d'erreurs consiste à identifier l'emplacement des instructions suspectes [Wong 2009a] afin d'aider l'utilisateur à comprendre pourquoi le programme a échoué, ce qui lui facilite la tâche de la correction d'erreurs. En BMC (Bounded Model Checking), la postcondition violée devient correcte, si on supprime l'instruction suspectée. En test, une instruction suspectée est apparue dans les tests erronés, elle est choisie selon certains critères. Lorsqu'un programme est non-conforme vis-à-vis de sa spécification ou dit erroné, l'utilisateur fouille la trace d'exécution d'un contre-exemple pour découvrir comment et où les erreurs sont introduites dans le code source. Face à la complexité d'un système et même si l'utilisateur est expérimenté, ce processus peut être difficile quand il est fait à la main. Cette recherche peut être menée à bien en passant par des techniques automatisées avancées qui imposent un minimum d'effort pour les développeurs afin de repérer les erreurs probables. Le but de ses techniques est de fournir un ensemble réduit d'instructions qui sont susceptibles d'être causes de l'échec dans le programme.

1.3 Aperçu de notre solution

Pour résoudre ce problème, nous avons proposé une approche [Bekkouche 2014, Bekkouche 2015b] à base de contraintes qui explore les chemins du CFG (Control Flow Graph) du programme à partir du contre-exemple. Cette approche calcule les sous-ensembles minimaux dont la modification peut permettre de restaurer la conformité du programme vis-à-vis de sa postcondition, pour le contre-exemple considéré.

Nous générons un système de contraintes pour les chemins du CFG où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces systèmes de contraintes, augmentés de la postcondition et du contre-exemple, des ensembles minima de correction (MCS - Minimal Correction Subset) de taille bornée. Le retrait d'un de ces ensembles produit un MSS (Maximal Satisfiable Subset) qui rend le système considéré satisfiable. Nous avons adapté pour cela un algorithme proposé par Liffiton et Sakallah afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques.

Plus formellement, la localisation d'erreurs utilise en entrée un programme erroné qui contredit sa postcondition, la postcondition violée, et un contre-exemple fourni par un outil de BMC (Bounded Model Checking). Elle permet de fournir des informations utiles pour trouver les erreurs potentielles dans le programme, ce qui permet au programmeur de comprendre l'origine de ses erreurs.

1.4 Principales contributions

L'idée de notre approche est de réduire le problème de la localisation d'erreurs vers celui qui consiste à calculer un ensemble minimal qui explique pourquoi un CSP (Constraint Satisfaction Problem) est infaisable. Le CSP représente l'union des contraintes du contre-exemple, du programme et de l'assertion violée. L'ensemble calculé peut être un MCS (Minimal Correction Subset) ou MUS (Minimal Unsatisfiable Subset). En général, tester la faisabilité d'un CSP sur un domaine fini est un problème NP-Complet (intraitable) [4], la classe des problèmes les plus difficiles de la classe NP. Cela veut dire, qu'expliquer l'infaisabilité dans un CSP est aussi dur, voire plus (on peut classer le problème comme NP-Difficile). `BugAssist` [Jose 2011c, Jose 2011a] est une méthode de localisation d'erreurs qui utilise un solveur Max-SAT pour calculer la fusion des MCSs de la formule Booléenne du programme en entier avec le contre-exemple. Elle devient inefficace pour les programmes qui contiennent des opérations arithmétiques sur les entiers. Notre approche travaille aussi à partir d'un contre-exemple pour calculer les MCSs. La contribution de notre approche par rapport à `BugAssist` peut se résumer dans les points suivants :

4. Si ce problème pouvait être résolu en temps polynomial, alors tous les problèmes NP-Complet le seraient aussi.

* Nous ne transformons pas la totalité du programme en un système de contraintes, mais nous utilisons le CFG du programme pour collecter les contraintes du chemin du contre-exemple et des chemins dérivés de ce dernier, en supposant qu'au plus k instructions conditionnelles sont susceptibles de contenir les erreurs.
* Nous calculons les MCSs uniquement sur le chemin du contre-exemple et les chemins déviés du contre-exemple permettant de satisfaire la postcondition ;
* Nous ne traduisons pas les instructions du programme en une formule SAT, mais plutôt en contraintes numériques qui vont être manipulées par des solveurs de contraintes ;
* Nous n'utilisons pas des solveurs MaxSAT comme boîtes noires, mais plutôt un algorithme générique pour calculer les MCSs par l'usage d'un solveur de contraintes.
* Nous bornons la taille des MCSs générés et le nombre de conditions déviées.
* Nous pouvons faire collaborer plusieurs solveurs durant le processus de localisation et prendre celui le plus performant selon la catégorie du CSP construit. Exemple, si le CSP du chemin détecté est du type linéaire sur les entiers, nous faisons appel à un solveur MIP (Mixed Integer Programming) ; s'il est non linéaire, nous utilisons un solveur CP (Constraint Programming) ou aussi MINLP (Mixed Integer Nonlinear Programming).

1.5 Organisation du Manuscrit

Cette thèse est divisée en quatre parties et contient 6 chapitres.

Le chapitre 1 présente l'introduction. Nous parlons dans ce chapitre de notre motivation ; nous expliquons le problème de localisation d'erreurs ; nous donnons un aperçu de notre solution et nous présentons nos principales contributions.

La partie I est consacrée à un positionnement de notre approche par rapport aux principales méthodes qui ont été proposées pour résoudre le problème de la localisation d'erreurs. Dans cette partie, le chapitre 2 parle des méthodes utilisées pour l'aide à la localisation des erreurs dans le cadre du test et de la vérification de programmes. Comme l'approche que nous proposons consiste essentiellement à rechercher des sous-ensembles de contraintes spécifiques dans un système de contraintes inconsistant, le chapitre 3 parle aussi des algorithmes qui ont été utilisés en recherche opérationnelle et en programmation par contraintes pour aider l'utilisateur à debugger un système de contraintes inconsistant.

La partie II est dédiée à notre approche pour localiser les erreurs, `LocFaults` (voir le chapitre 4). Elle présente notre nouvelle approche basée sur la CP pour l'aide à la localisation d'erreurs dans un programme pour lequel un contre-exemple a été trouvé ; nous détaillons notre méthode. Elle présente également une exploration de `LocFaults` pour le traitement des boucles erronées, particulièrement sur

le traitement des boucles avec le bug Off-by-one[5].

La partie III présente notre évaluation expérimentale (voir le chapitre 5). Elle décrit l'implémentation de notre outil fondé sur les contraintes pour l'aide à la localisation d'erreurs dans les programmes JAVA. Elle présente également notre étude expérimentale sur un ensemble de benchmarks académiques et réalistes. Nous présentons aussi une comparaison avec l'outil concurrent `BugAssist`, un système de l'état de l'art de la localisation d'erreurs.

La partie III contient le chapitre qui clôture ce manuscrit avec des conclusions et nos travaux futurs, voir le chapitre 6.

5. Il apparaît régulièrement en programmation lorsqu'une boucle s'itère trop ou trop peu.

Première partie

Revue de la littérature

Cette partie a pour but de rassembler les techniques importantes qui ont été proposées pour résoudre le problème de la localisation d'erreurs. Dans le premier chapitre, nous présentons les méthodes utilisées pour l'aide à la localisation d'erreurs en test et vérification de programmes. Dans le chapitre 2, nous parlons des méthodes pour connaître les causes des erreurs et diagnostiquer un système de contraintes infaisable ; nous nous intéressons particulièrement, dans ce chapitre, à présenter les algorithmes les plus connus qui calculent les MCSs (Minimal Correction Subsets) et MUSs (Minimal Unsatisfiable Subsets). Le chapitre 3 rappelle aussi les concepts de base pour la localisation d'erreurs sur les systèmes de contraintes numériques, ces concepts sont utiles pour la compréhension du reste du manuscrit.

Chapitre 2

Méthodes d'aide à la localisation des erreurs utilisées en test et vérification de programmes

"Je ne crains pas à la montée de l'intelligence artificielle comme Stephen Hawking, Bill Gates et Elon Musk; ce qui me fait peur est le logiciel fragile sur lequel repose la société." – Grady Booch

Sommaire

2.1 Introduction

La recherche des bugs dans un programme porte sur l'analyse de traces d'exécution. Cette analyse consiste à collecter des données lors de l'exécution du programme afin de déterminer la localisation probable du bug.

Plusieurs travaux ont été menés pour l'aide à la localisation d'erreurs dans un programme impératif. Nous pouvons les classer en fonction de leurs caractéristiques communes. Ce chapitre a pour but de présenter les méthodes principalement utilisées en test et vérification de programmes. Nous commençons par présenter les approches proposées en test, ensuite celles proposées en Model Checking.

2.2 La localisation d'erreurs en test

En développement du logiciel piloté par les tests [Gotlieb 1998], les tests unitaires jouent un rôle important pour assurer la qualité des logiciels. Les cas de tests

peuvent être utilisés pour la localisation des erreurs [Xuan 2014]. En effet, la localisation d'erreurs en test consiste à comparer deux types de traces d'exécution : les exécutions correctes (dites positives) et les exécutions erronées (dites négatives). Pour mesurer la suspicion de chaque instruction dans le programme, ces méthodes utilisent des métriques pour calculer une distance entre la ligne d'instruction et le bug, en exploitant les occurrences des instructions apparaissant dans les traces correctes et erronées. Soit un ensemble de cas de test T pour un programme erroné P et un cas de test t dans T, ces méthodes associent deux types d'informations pour chaque exécution du programme P pour l'entrée t :

- pass (notée P), dans le cas où la sortie de t sur le programme est correcte ;
- fail (notée F), dans le cas où la sortie de t sur le programme est fausse.

Le code couvert pour t représente l'ensemble des instructions exécutées. Pour localiser automatiquement les erreurs sur base de tests automatisés, les données suivantes sont collectées pour chaque instruction dans le programme :

- e_f : le nombre de tests exécutant l'instruction traitée qui ont échoué ;
- e_p : le nombre de tests exécutant l'instruction traitée qui ont réussi ;
- n_f : le nombre de tests n'exécutant pas l'instruction traitée qui ont échoué ;
- n_p : le nombre de tests n'exécutant pas l'instruction traitée qui ont réussi.

```
1  void mid (int a, int b, int c) {
2    int m;
3    m=c;
4    if (b<c){
5         if (a<b){
6              m=b;
7         }
8         else if (a<c){
9           m=b; // error, the statement should be m=a;
10        }
11   }
12   else{
13        if (a>b){
14          m=b;
15        }
16        else if (a>c){
17          m=a;
18        }
19   }
20 }
```

FIGURE 2.1 – Le programme Mid

Pour illustrer comment ces informations sont utilisées, nous utilisons un programme simple (voir sur la figure 2.1), et une suite de test. Ce programme (nommé `Mid`) utilisé dans [Jones 2005] prend trois entiers (a, b, c) en entrée et retourne la valeur médiane. Le programme contient une erreur sur la ligne 9, cette ligne devrait se lire "$m = a$". Les cas de tests utilisés sont $t_1 = \{a = 3, b = 3, c = 5\}$, $t_2 = \{a = 1, b = 2, c = 3\}$, $t_3 = \{a = 3, b = 2, c = 1\}$, $t_4 = \{a = 5, b = 5, c = 5\}$, $t_5 = \{a = 5, b = 3, c = 4\}$, $t_6 = \{a = 2, b = 1, c = 3\}$. L'ensemble des données collectées, pour chaque ligne dans le programme `Mid`, sont affichées dans la table 2.1. Dans cet exemple, la ligne 5 est exécutée par les cas de tests t_1, t_2, t_5, t_6. Il y a un cas de test qui a échoué (e_f), trois ont réussi (e_p), le seul cas de test échoué a exécuté la ligne (n_f) et deux tests réussis n'exécutent pas cette ligne (n_p).

	t_1	t_2	t_3	t_4	t_5	t_6	e_f	e_p	n_f	n_p
Ligne 3	✓	✓	✓	✓	✓	✓	1	5	0	0
Ligne 4	✓	✓	✓	✓	✓	✓	1	5	0	0
Ligne 5	✓	✓			✓	✓	1	3	0	2
Ligne 6		✓					0	1	1	4
Ligne 8	✓				✓	✓	1	2	0	3
Ligne 9	✓					✓	1	1	0	4
Ligne 13			✓	✓			0	2	1	3
Ligne 14			✓	✓			0	2	1	3
Ligne 16			✓				0	1	1	4
Ligne 17				✓			0	1	1	4
Résultats des tests	P	P	P	P	P	F				

TABLE 2.1 – Un exemple de collecte de données

Dans la suite de cette section, nous allons présenter des outils et approches permettant la localisation d'erreurs en test.

Tarantula De nombreux systèmes ont été développés la localisation d'erreurs en test, le plus célèbre est `Tarantula` [Jones 2001, Jones 2002]. Il consiste à colorer les instructions selon leur participation dans le test du programme erroné. L'approche discrète de coloriage utilise une simple technique pour colorer les instructions :

- Les instructions exécutées seulement durant des exécutions fausses sont colorées en rouge ;
- Les instructions exécutées seulement durant des exécutions correctes sont colorées en vert ;
- Les instructions sont colorées en jaune dans le cas où elles sont exécutées durant des exécutions correctes et fausses.

Cette approche simple ne permet pas de fournir beaucoup d'aides utiles au programmeur sur l'emplacement des erreurs dans le programme erroné. Pour mieux assister le programmeur dans la tâche de localisation des erreurs, l'approche `Tarantula` propose un coloriage non-discret (continu) pour colorer les instructions qui participent aux exécutions correctes et fausses. Voici le mode de coloriage employé par la méthode :

- Les instructions sont plus vertes si le pourcentage des exécutions correctes couvrant ces instructions est plus élevé ;
- Les instructions sont plus rouges si le pourcentage des exécutions fausses couvrant ces instructions est plus élevé ;
- Les instructions sont jaunes dans le cas où les pourcentages des exécutions correctes et fausses couvrant ces instructions sont à peu près égales.

L'idée de cette méthode est que les instructions représentant un danger (avec un degré de suspicion élevé) pour la correction du programme sont principalement cou-

vertes par des exécutions fausses. Les instructions principalement couvertes par des exécutions correctes sont plus sûres et ont un degré de suspicion faible. Les instructions passées par un mélange d'exécutions correctes et fausses sont plus-ou-moins neutres vis-à-vis les erreurs dans le programme. `Tarantula` utilise le spectre des couleurs du rouge au jaune pour colorer chaque instruction dans le programme sous test. Le spectre d'une instruction, s, (noté hue(s)) est calculé à l'aide de l'équation suivante :

$$hue(s) = \frac{\frac{e_p(s)}{totalpassed}}{\frac{e_p(s)}{totalpassed} + \frac{e_f(s)}{totalfailed}} \tag{2.1}$$

Dans l'équation 2.1 : $e_p(s)$ et $e_f(s)$ représentent respectivement le nombre de tests réussis et échoués exécutant l'instruction s, $totalpassed$ et $totalfailed$ sont le nombre total de tests réussis et échoués. Les valeurs calculées peuvent être utilisées sans visualisation [Jones 2005]. $hue(s)$ est utilisé pour exprimer le degré de suspicion de l'instruction s. L'ensemble des valeurs réelles retourné $hue(s)$ varie entre 0 et 1 : 0 pour exprimer que l'instruction est la plus suspecte, 1 veut dire que l'instruction a le degré de suspicion le plus faible. Pour exprimer la valeur de suspicion d'une manière plus intuitive (la valeur augmente avec la suspicion), `Tarantula` utilise la fonction suivante :

$$suspiciousness(s) = 1 - hue(s) = \frac{\frac{e_f(s)}{totalfailed}}{\frac{e_p(s)}{totalpassed} + \frac{e_f(s)}{totalfailed}} \tag{2.2}$$

`Tarantula` ordonne les instructions selon les scores de suspicion calculés à l'aide de l'équation 2.2. Par exemple, dans le programme `Mid` avec les six cas de test, l'instruction à la ligne 9 est la première instruction que le programmeur doit inspecter avec un score de $0,83$ ($suspiciousness$(ligne 9) $= 0,83$). Le tableau 2.2 présente le score de suspicion et le rang de chaque instruction dans le programme `Mid`.

Ligne	3	4	5	6	8	9	13	14	16	17
Score de suspicion	0.5	0.5	0.63	0.0	0.71	0.83	0.0	0.0	0.0	0.0
Rang	7	7	3	13	2	1	13	13	13	13

TABLE 2.2 – Le score de suspicion et le rang de chaque instruction dans le programme `Mid` à l'aide de l'équation 2.2 [Jones 2005].

Pinpoint `Pinpoint` [Chen 2002] est une approche pour analyser la cause d'erreurs dans la plate-forme J2EE. Elle est développée dans le cadre du projet de "the Recovery Oriented Computing (ROC)" [Patterson 2002], et destinée aux grands services dynamiques d'Internet , tels que les services Web-mail et les moteurs de recherche. `Pinpoint` se compose de trois parties : une couche de communication qui trace les

demandes des clients, un détecteur de défaillance qui utilise l'instrumentation du trafic renifleurs (traffic-sniffing) et middleware, et un moteur d'analyse de données pour identifier le composant ou les combinaisons de composants susceptibles d'être à l'origine du bug dans les traces obtenues. `Pinpoint` a été évalué en injectant les erreurs dans divers composants de l'application Java PetStore (exemple d'applications pour J2EE) sur une série d'exécutions. `Pinpoint` arrive à identifier les composants défectueux avec une grande précision et produit peu de faux positifs (false-positives).

Delta Debugging Lorsque le test échoue, il est possible de démarrer le débogage. Cependant, cela n'est pas une bonne idée, car l'exécution contient souvent trop d'étapes qui est difficile à étudier. L'algorithme `Delta Debugging` de Andreas Zeller [Zeller 2002] se base sur le test pour simplifier et isoler l'échec dans cette trace d'exécution. La simplification permet de réduire la taille de certains cas de test erronés à un cas de test minimal qui produit encore de l'échec, en utilisant la technique de diviser pour régner. Il arrête quand un cas de test minimal est atteint, où la suppression de toute entité d'entrée corrige l'échec. L'isolation consiste à trouver la différence entre un cas de test erroné et correct.

Pour illustrer la simplification, voyons un exemple[1] qui implémente l'algorithme de Quicksort (voir fig. 2.2) et le cas de test suivant : $\{4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3\}$. La sortie obtenue en exécutant le programme sur ce cas de test est : $\{1, 3, 2, \mathbf{5}, 3, 4, 4, 6, 7, 8, 9, 10, 11, 12, 14, 17, 19, 21\}$ qui est fausse, donc le test a échoué. Pour trouver le bug, nous devons analyser la trace d'exécution de ce cas de test. Le nombre d'étapes d'exécution est de 670, ce qui est difficile à suivre. Voici les étapes d'exécution de l'algorithme `Delta Debugging` pour réduire cette trace :

1. Le cas de test est divisé en deux. La première entrée de l'algorithme est $\{4, 12, 9, 14, 3, 10, 17, 11, 8\}$, qui produit encore de l'échec : la sortie du programme est $\{3, 12, 4, 8, 9, 10, 11, 14, 17\}$.
2. La deuxième entrée est $\{4, 12, 9, 14\}$ pour laquelle la sortie est correcte : $\{4, 9, 12, 14\}$.
3. L'algorithme essaye donc, dans cette étape, l'autre moitié : $\{3, 10, 17, 11, 8\}$. La sortie obtenue est aussi correcte.
4. En supprimant les deux derniers numéros de l'entrée à l'étape (1), nous obtenons le cas de test suivant $\{4, 12, 9, 14, 3, 10, 17\}$. Il produit une sortie erronée.
5. Arrêt du processus de la simplification, l'algorithme `Delta Debugging` trouve un cas de test minimal où enlever tout élément d'entrée permet d'avoir une sortie correcte. Pour ce cas de test, le nombre d'étapes d'exécution est de 192, ce qui représente seulement 30% de la trace originale.

1. L'exemple est pris à partir du lien : https://dzone.com/articles/debugging-step-step-delta

```
1  public class QuickSort
2  {
3    public static void quicksort(int numbers[], int low, int high) {
4        int i = low, j = high;
5        // Get the pivot element from the middle of the list
6        int pivot = numbers[low + (high-low)/2];
7
8        // Divide into two lists
9        while (i <= j) {
10         while (numbers[i] < pivot) {
11           i++;
12         }
13         while (numbers[j] > pivot) {
14           j--;
15         }
16         if (i <= j) {
17           exchange(numbers, i, j);
18         }
19         i++;
20         j--;
21       }
22       // Recursion
23       if (low < j)
24         quicksort(numbers, low, j);  //low
25       if (i < high)
26         quicksort(numbers, i, high);
27     }
28
29     private static void exchange(int numbers[], int i, int j) {
30       int temp = numbers[i];
31       numbers[i] = numbers[j];
32       numbers[j] = temp;
33       int k = 1;
34     }
35
36   public static void main(String argv[])
37    {
38     int A[] = {4, 12, 9, 14, 3, 10, 17, 11, 8, 7, 4, 1, 6, 19, 5, 21, 2, 3};
39
40 //     minimized failure-induced input
41 //     int A[] = {3, 5, 2, 3};
42
43     quicksort(A, 0, A.length - 1);
44
45     for (int i=0 ; i < A.length ; i++) System.out.print(A[i] + " ");
46     System.out.println();
47    }
```

FIGURE 2.2 – Quicksort avec bug

WHITHER Renieris et Reiss [Renieris 2003] proposent une approche qui utilise des séries de tests correctes et des séries erronées. Cette méthode suppose l'existence d'un cas de test erroné et un grand nombre de cas de test corrects. Il sélectionne selon un critère de distance le cas de test correct qui ressemble le plus au cas de test erroné, compare les spectres correspondant à ces deux cas de test, et produit un rapport de parties "suspectes" du programme. Les auteurs ont développé l'outil `WHITHER` qui implémente leur approche. Les résultats expérimentaux montrent que la qualité de localisation de cette approche dépend fortement de la qualité des cas de tests utilisés.

Approche de Liblit et al. Liblit et al. [Liblit 2005] proposent un algorithme de débogage statistique pour isoler plusieurs bugs dans des systèmes logiciels complexes. Il consiste à analyser les prédicats à la place des entités exécutées (lignes, blocs, classes, ...). La technique travaille en séparant les effets de différents bugs et identifie les prédicats qui sont associés avec des bugs individuels. Ces prédicats révèlent à la

fois les circonstances dans lesquelles les bugs se produisent ainsi que les fréquences des modes de défaillance, ce qui rend plus facile de prioriser les efforts de débogage. Soit B désigne un bug. L'approche utilise un profil de bug (noté β). Soit un ensemble d'exécutions fautives qui partagent B comme la cause de l'échec. Un prédicat P est un prédicat de bug B si chaque fois $R(P) = 1$ [2] alors il est statistiquement probable que $R \in \beta$. Cette approche sélectionne un petit sous-ensemble S de l'ensemble de tous les prédicats instrumentés P, tel que S a des prédicats de tous les bugs. Pour finir, elle classe les prédicats dans S du plus au moins important, en utilisant des métriques qui permettent de trier les prédicats en fonction de leur implication dans les tests erronés.

Approche de Guo et al. À partir d'une exécution erronée, une approche de localisation d'erreurs procède souvent en comparant le cas de test erroné avec un cas de test réussi. Une question importante ici est le choix de l'exécution réussie pour une telle comparaison. Pour cela, Guo et al. [Guo 2006] proposent une approche de métrique de différence basée sur les flots de contrôle. Étant donné une exécution erronée π_f et un ensemble de cas de test réussis S, ils choisissent les π_s de cas de test réussis de S dont la trace d'exécution est plus proche à π_f en termes de différence métrique. Un rapport de bogue est alors généré par le retour de la différence entre π_f et π_s.

SOBER Liu et al. [Liu 2006] proposent une approche statistique, nommée SOBER, pour localiser les erreurs, sans connaissance préalable de la sémantique du programme. Cette approche classe les prédicats suspects. Un prédicat P peut être évalué pour être vrai plus d'une fois dans une exécution. $\pi(P)$ est la probabilité que P soit évalué à vrai dans chaque exécution. $\pi(P) = \frac{n(f)}{n(t)+n(f)}$ tel que $n(t)$ est le nombre de fois que P est évalué pour être vrai dans une exécution spécifique et $n(f)$ est le nombre de fois que P est évalué à faux. Si la distribution de $\pi(P)$ pour une exécution erronée est significativement différente de celle de $\pi(P)$ dans une exécution réussie, alors P est liée à une erreur [Wong 2009a].

AMPLE `AMPLE` (Analyzing Method Patterns to Locate Errors) [Abreu 2007] est un système d'identification des classes défectueuses dans un logiciel orienté objet, avec la classe comme unité d'étude. Il fonctionne en comparant les séquences d'appels entre les méthodes exécutées pendant les tests corrects et erronés. Comme résultat, AMPLE présente un classement des classes, les classes au sommet sont celles qui sont susceptibles d'être responsables de l'échec dans le programme.

Approche de Xuan et al. Quand un cas de test erroné est généré, une seule assertion est concernée. Xuan et al. [Xuan 2014] affirme que l'exécution de toutes

2. Si P est observé pour être vrai au moins une fois pendant l'exécution R alors $R(P) = 1$, sinon $R(P) = 0$.

```
try {
   /* assertion */
}
catch (java.lang.Throwable throwable) {
   /* do nothing */
}
```

FIGURE 2.3 – Une structure pour transformer une assertion en une instruction régulière de cas de test.

les assertions peut améliorer l'efficacité de la localisation d'erreurs, car son efficacité dépend de la qualité de l'assertion à vérifier. Pour cela, à partir d'un cas de test erronés, l'approche de Xuan et al. génère des cas de tests erronés supplémentaires pour exécuter toutes les assertions. Cette approche propose un nouveau concept de purification de cas de tests pour améliorer les techniques de localisation d'erreurs existantes, comme `Tarantula`. Cette approche se déroule en trois étapes :

1. La première consiste à générer un ensemble de cas de tests pour chaque cas de test erroné. À partir d'un cas de test erroné i avec k assertions, cette approche commence par créer k exemplaires pour le cas de test. Ensuite, pour une copie i du cas de test, elle entoure toutes les assertions sauf la i^{ime} assertion avec une structure *try/catch* (voir fig. 2.3) pour ignorer son exception. Après cela, elle compile et exécute tous les cas de test générés. Finalement, elle enregistre les cas de test erronés et les positions qui interrompent l'exécution (une erreur inattendue).
2. L'objectif de la deuxième étape est de générer des cas de tests purifiés avant de collecter leurs spectres (la trace d'exécution pour chaque cas de test purifié). Après cela, elle compile et exécute tous les tests purifiés. Enfin, elle collecte la trace d'exécution pour chaque cas de test purifié.
3. La troisième étape consiste à calculer le degré de suspicion et classer les instructions en utilisant une technique statistique existante de localisation d'erreurs (exemple `Tarantula`) avec les spectres obtenues de la phase précédente.

2.3 Conclusion

Les techniques de localisation d'erreurs sur la base de tests exploitent la disponibilité d'une bonne suite de cas tests. Elles utilisent différentes métriques pour établir un classement des instructions suspectes détectés lors de l'exécution d'une batterie de test. Ces approches présentent l'avantage qu'elles sont simples. Leur faiblesse est qu'elles nécessitent un grand nombre de cas de test. La qualité de l'ensemble de cas de tests employé a un impacte sur leur l'efficacité. Aussi, pour ces techniques, les dépendances entre les instructions ne sont pas prises en compte. Un autre point critique dans les approche statistiques réside dans le fait qu'elle requiert un oracle qui permet de décider si le résultat d'un test est juste ou non.

Quand une erreur détectée par un **Model Checker** n'est pas couverte par un cas de test, les techniques à base de test sont peu utilisées [Groce 2005]. Nous présentons dans la section suivante les approches de localisation d'erreurs en **Model Checking**, c'est-à-dire un cadre où les seuls prérequis sont un programme, une postcondition ou une assertion qui doit être vérifiée, et éventuellement une précondition.

2.4 La localisation d'erreurs en Model Checking

Le Model Checking a été introduit par Clarke et Emerson [Clarke 1982]. L'idée de base est de déterminer si un programme est conforme vis-à-vis d'une spécification, en explorant exhaustivement les états accessibles du programme. Si le programme est erroné (le programme est non-conforme vis-à-vis de sa spécification) le Model Checker génère un contre-exemple, et une trace d'exécution conduisant à un état pour lequel la propriété est violée [D'silva 2008]. Un Model Checker peut être utilisé non seulement pour détecter les erreurs, mais aussi pour expliquer et aider à localiser ces erreurs. Cette section explore les approches de localisation d'erreurs en Model Checking.

Approche de Bal et al. Bal et al.[Ball 2003] utilisent plusieurs appels à un Model Checker et comparent les contre-exemples obtenus avec une trace d'exécution correcte. Les transitions qui ne figurent pas dans la trace correcte sont signalées comme une possible cause de l'erreur. Ils ont implanté leur algorithme dans le contexte de SLAM, un model checker qui vérifie les propriétés de sécurité temporelles dans un programme C.

Explain Plus récemment, des approches basées sur la dérivation de traces correctes ont été introduites dans un système nommé `Explain`[Groce 2004, Groce 2006]. Ce système étend CBMC [3], un vérificateur de modèle borné pour les programmes écrits en ANSI C. Il est basé sur l'approche contrefactuelle de causalité mise en place par David Lewis [Lewis 1973]. Pour un programme P, le processus se déroule comme suit :

1. L'outil de bounded model checking `CBMC` [4] utilise un dépliage des boucles et la forme SSA (Static Single Assignment) pour transformer le programme erroné et sa spécification en un problème SAT, noté S. Les instanciations satisfaisantes de S sont les contre-exemples ;
2. `CBMC` utilise un solveur SAT pour trouver un contre-exemple ;
3. L'outil `Explain` produit une formule booléenne, S'. Les assignations satisfaisantes de S' sont les exécutions de P qui ne violent pas la postcondition.

3. http://www.cs.cmu.edu/~modelcheck/cbmc/
4. http://www.cprover.org/cbmc/

`Explain` étend S' avec les contraintes pour représenter le problème d'optimisation qui consiste à trouver une assignation qui est aussi similaire que possible au contre-exemple (une distance métrique sur les exécutions du programme est utilisée) ;

4. Il utilise un solveur pseudo-booléen pour rechercher l'exécution correcte la plus proche au contre-exemple ;
5. Les différences entre les traces réussies et le contre-exemple sont calculées ;
6. Avant de présenter les différences calculées à l'utilisateur, une étape de slicing est appliquée afin de les réduire.

Approche de Griesmayer et al. Dans [Griesmayer 2006, Griesmayer 2007], les auteurs proposent une approche automatique pour la localisation d'erreurs dans les programmes C. Elle suppose que le programme contient une erreur, et l'existence d'un contre-exemple qui contredit la spécification. Le contre-exemple est ensuite utilisé pour créer une version étendue du programme. Elle introduit ensuite dans le programme des prédicats anormaux pour chaque composant ; tel que si le prédicat anormal d'un composant est vrai, alors le composant fonctionne anormalement. Par conséquent, le comportement d'origine de la composante est suspendu et remplacé par une nouvelle entrée. Pour réduire le nombre d'erreurs potentielles, le processus est redémarré pour différents contre-exemples. Les auteurs calculent l'intersection des ensembles d'instructions suspectes. Cependant, cette approche souffre de deux problèmes majeurs :

- Elle permet de modifier n'importe quelle expression, l'espace de recherche peut ainsi être très grand ;
- Elle peut renvoyer beaucoup de faux diagnostics totalement absurdes car toute modification d'expression est possible (par exemple, changer la dernière affectation d'une fonction pour renvoyer le résultat attendu).

Approche de Zhang et al. Pour remédier aux inconvénients de [Griesmayer 2006, Griesmayer 2007], Zhang et al [Zhang 2006] proposent de modifier uniquement les prédicats de flux de contrôle. L'intuition de cette approche est qu'à travers un switch des résultats d'un prédicat et la modification du flot de contrôle, l'état de programme peut non seulement être modifié à peu de frais, mais qu'en plus, il est souvent possible d'arriver à un état succès. Liu et al [Liu 2010] généralisent cette approche en permettant la modification de plusieurs prédicats. Ils proposent également une étude théorique d'un algorithme de débogage pour les erreurs *RHS*, c'est à dire les erreurs situées dans les prédicats de contrôle et la partie droite des affectations.

Approche de Ilan Beer et al. Dans [Beer 2009], les auteurs abordent le problème de l'analyse de la trace d'un contre-exemple et de l'identification de l'erreur dans le cadre des systèmes de vérification formelle du hardware. Ils utilisent pour

cela la notion de causalité introduite par Halpern et Pearl pour définir formellement une série de causes de violation de la spécification par un contre-exemple.

BugAssist Récemment, Manu Jose et Rupak Majumdar [Jose 2011b, Jose 2011d] ont abordé ce problème différemment : ils ont introduit un nouvel algorithme qui utilise un solveur partiel MaxSAT pour calculer le nombre maximum des clauses d'une formule booléenne qui peut être satisfaite par une affectation. Leur algorithme fonctionne à partir de l'entrée de test (le contre-exemple) en trois étapes :

1. Il commence par construire une formule booléenne de la trace du programme qui correspond au contre-exemple (notée F). Elle est sous forme normale conjonctive et satisfiable ssi l'exécution du programme est réalisable. Tel que toute affectation satisfiable de la formule correspond à une séquence d'états dans une exécution du programme ;

2. Il étend ensuite la formule de trace F à une formule fausse F' en imposant la postcondition à vraie (la formule F' est insatisfiable car la trace correspond à un contre-exemple qui viole la postcondition). En d'autres termes, il ajoute à la formule de trace F les contraintes garantissant que l'état initial satisfait les valeurs du cas de test erroné, et que les états finaux satisfont la postcondition qui a été violée par le test du programme ;

3. Finalement, il utilise le solveur partiel MaxSAT pour calculer le nombre maximum de clauses pouvant être satisfaites dans F' et prend le complément de cet ensemble comme une cause potentielle des erreurs. En d'autres termes, ils calculent le complément d'un MSS (Maximal Satisfiable Subset). Précisément, il marque les contraintes sur les entrées ainsi que la postcondition comme clauses dures et le reste des contraintes comme clauses molles. Il donne la formule étendue obtenue au solveur partiel MaxSAT pour trouver le nombre maximum de clauses molles qui peuvent être satisfaites par une affectation qui satisfait toutes les clauses dures. Puisqu'il peut y avoir plusieurs ensembles minimaux de clauses qui peuvent être trouvés dans cette formule, il énumère donc chaque ensemble minimal comme localisation probable. Il affiche à l'utilisateur l'ensemble qui fusionne toutes les localisations trouvées.

Manu Jose et Rupak Majumdar [Jose 2011b, Jose 2011d] ont implanté leur algorithme dans un outil appelé `BugAssist` qui utilise `CBMC`. `BugAssist` prend en entrée un programme C avec une assertion, et un ensemble de cas de test erronés, et renvoie un ensemble d'instructions du programme dont le remplacement peut éliminer les erreurs. Pour construire la formule de trace, il utilise l'outil de bounded model checking CBMC [5]. Pour calculer l'ensemble maximal des clauses satisfaites, il utilise des solveurs MaxSAT scalables [Marques-Sila 2011], comme MSUnCore [Marques-Silva 2009].

5. http ://www.cprover.org/cbmc/

SNIPER Si-Mohamed Lamraoui et Shin Nakajima [Lamraoui 2014] ont aussi développé récemment un outil nommé `SNIPER` qui calcule les MSSs d'une formule $\psi = EI \wedge TF \wedge AS$ où EI encode les valeurs d'entrée erronées, TF est une formule qui représente tous les chemins du programme, et AS correspond à l'assertion qui est violée. Les MCSs sont obtenus en prenant le complément des MSSs calculés. L'implémentation est basée sur la représentation intermédiaire `LLVM` et le solveur SMT `Yices`. L'implémentation actuelle est toutefois beaucoup plus lente que `BugAssist`. Les auteurs ont comparé SNIPER avec `BugAssist` sur le benchmark TCAS de la suite de test de Siemens. `SNIPER` a identifié 5% d'erreurs en plus que `BugAssist` mais a nécessité beaucoup plus de temps d'exécution que `BugAssist` sur ce benchmark. Cette approche suppose l'existence d'un cas de test qui déclenche tous les défauts dans le programme.

Pour cette thèse, l'approche que nous avons proposée est inspirée des travaux de Manu Jose et Rupak Majumdar. Les principales différences sont :

1. Nous ne transformons pas tout le programme en un système de contraintes mais nous utilisons le graphe de flot de contrôle pour collecter les contraintes qui correspondent aux instructions du chemin d'un contre exemple.
2. Nous n'utilisons pas des algorithmes basés sur `MaxSAT` mais des algorithmes plus généraux qui permettent plus facilement de traiter des contraintes numériques.

2.5 Conclusion

Les approches de localisation d'erreurs en test ont obtenus de très bons résultats avec la technique statistique `Tarantula` qui arrive presque toujours à localiser l'erreur mais en pratique `Tarantula` n'est pas facile à utiliser. Cependant en pratique, il faut disposer soit d'une spécification formelle complète, soit d'un oracle complet ; les personnes qui font l'effort de développer une spécification complète préfèrent faire de la preuve qui offre de bien meilleures garanties que le test ; et ceux qui n'ont pas de spécification, n'ont souvent pas d'oracle.

L'intérêt des approches de Model Checking comme `BugAssist` ou comme la notre, c'est qu'il ne faut qu'une postcondition pour les mettre en oeuvre. L'approche que nous avons proposée est proche de `BugAssist` et de `SNIPER`.

Chapitre 3

Calcul MCS et MUS dans un système de contraintes inconsistant

"Pour résoudre numériquement certains problèmes, il faudrait qu'un ordinateur aussi grand que l'Univers travaille pendant un temps supérieur à l'âge de l'Univers ! Ces problèmes sont malgé tout théoriquement résolubles." – Larry Stockmeyer et Ashok Chandra

Sommaire

3.1 Introduction

En recherche opérationnelle et en programmation par contraintes, différents algorithmes ont été proposés pour aider l'utilisateur à debugger un système de contraintes inconsistant. Lorsqu'on recherche des informations utiles pour la localisation d'erreurs sur les systèmes de contraintes numériques, nous pouvons nous intéresser à deux types d'informations :

1. Combien de contraintes dans un ensemble de contraintes insatisfiables peuvent être satisfaites ?
2. Quelle partie est inconsistante dans le système de contraintes ?

Pour analyser l'inconsistance dans un systèmes de contraintes, les sous-ensembles MCSs, MSSs, MUSs/IISs sont des outils essentiels. Dans ce chapitre, nous nous intéressons à présenter les principaux algorithmes cherchant à répondre à ces questions [1]. Mais avant, nous allons définir plus formellement les notions de MUS, MSS et MCS.

3.2 Définitions

À l'aide des définitions rappelées dans [Liffiton 2008], nous allons définir ce qu'est un MUS, un MSS et un MCS.

Soit C un ensemble de contraintes :
Un IIS (Irreducible Inconsistent Set) ou MUS (Minimal Unsatisfiable Subset) est un sous-ensemble de contraintes infaisable de C, et tous ses sous-ensembles stricts sont faisables. « An IIS has this property : it is itself infeasible, but any proper subset is feasible » (Chinneck [Chinneck 2008], P.93). « an irreducible infeasible subset (IIS) of the constraints, i.e. a (small) subset of constraints that is itself infeasible, but becomes feasible if one or more constraints is removed »(Chinneck [Chinneck 2008], Page.4).

$$M \subseteq C \text{ est un MUS} \Leftrightarrow M \text{ est UNSAT}$$
$$\text{et } \forall c \in M : M \setminus \{c\} \text{ est SAT.}$$

La notion de MSS (Maximal Satisfiable Subset) est une généralisation de MaxSAT / MaxCSP où l'on considère la maximalité au lieu de la cardinalité maximale :

$$M \subseteq C \text{ est un MSS} \Leftrightarrow M \text{ est SAT}$$
$$\text{et } \forall c \in C \setminus M : M \cup \{c\} \text{ est UNSAT.}$$

Cette définition est très proche de celle des IISs (Irreducible Inconsistent Subsystems) utilisés en recherche opérationnelle [Chinneck 1996, Chinneck 2001, Chinneck 2008].
Une MCS (Minimal Correction Set) est un sous-ensemble de contraintes de C dont le retrait produit un ensemble de contraintes faisables (il permet de "corriger"

1. Pour une présentation plus détaillée voirhttp://users.polytech.unice.fr/~rueher/Publis/Talk_NII_2013-11-06.pdf

l'infaisabilité). En outre, il est minimal dans dans le sens où n'importe quel sous-ensemble ne satisfait pas cette propriété. Les MCSs (Minimal Correction Set) sont des compléments des MSSs (le retrait d'un MCS à C produit un MSS car on "corrige" l'infaisabilité) :

$$M \subseteq C \text{ est un MCS} \Leftrightarrow C \setminus M \text{ est SAT}$$
$$\text{et } \forall c \in M : (C \setminus M) \cup \{c\} \text{ est UNSAT.}$$

3.2.1 Relation de dualité entre MUSs et MCSs

Il existe une dualité entre l'ensemble des MUSs et des MCSs [Birnbaum 2003, Liffiton 2008] : informellement, l'ensemble des MCSs est équivalent aux ensembles couvrants irréductibles[2] des MUSs ; et l'ensemble des MUS est équivalent aux ensembles couvrants irréductibles des MCSs. Soit un ensemble de contraintes C :

1. Un sous-ensemble M de C est un MCS ssi M est un ensemble couvrant minimal des MUS de C ;

2. Un sous-ensemble M de C est un MUS ssi M est un ensemble couvrant minimal des MCS de C ;

Au niveau intuitif, il est aisé de comprendre qu'un MCS doit au moins retirer une contrainte de chaque MUS. Et comme un MUS peut être rendu satisfiable en retirant n'importe laquelle de ses contraintes, chaque MCS doit au moins contenir une contrainte de chaque MUS. Cette dualité est aussi intéressante pour notre problématique car elle montre que les réponses aux deux questions posées en 3.1 sont étroitement liées.

Exemple Soit un système de contraintes $C = \{c_1, c_2, ..., c_{16}\}$(voir la figure 3.1).
Ce système de contraintes contient cinq MUSs : $\Sigma_{MUS} = \{S_1, S_2, S_3, S_4, S_5\}$.
$S_1 = \{c_3, c_5, c_9\}$, $S_2 = \{c_5, c_6, c_{11}, c_{14}\}$, $S_3 = \{c_6, c_{10}, c_{12}, c_{13}\}$, $S_4 = \{c_7, c_{11}\}$, $S_5 = \{c_4, c_{11}\}$
A partir de l'ensemble Σ_{MUS}, on peut calculer :

- L'ensemble qui contient tous les MCSs de cardinalité minimum : Il y a exactement douze ($|S_1| \times |S_3|$) dont la cartinalité est trois. Exemple, l'ensemble $\{c_3, c_{11}, c_{13}\}$.
- Les MCSs : $\{c_4, c_7, c_9, c_{10}, c_{14}\}$, $\{c_4, c_5, c_{11}, c_{13}\}$, tous les MCSs de cardinalité minimum, ...etc.

2. Soit Σ un ensemble d'ensemble et D l'union des éléments de Σ. On rappelle que H est un ensemble couvrant de Σ si $H \subseteq D$ et $\forall S \in \Sigma : H \cup S \neq \emptyset$. H est irréductible (ou minimal) si aucun élément ne peut être retiré de H sans que celui-ci ne perde sa propriété d'ensemble couvrant.

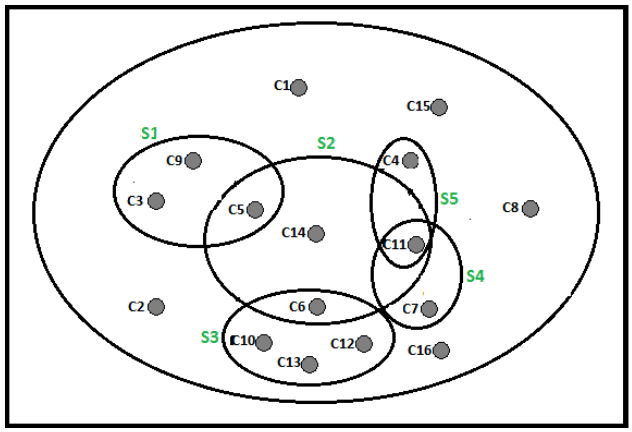


FIGURE 3.1 – Un système de contraintes avec cinq MUSs

3.3 Algorithmes pour calculer un seul MCS

3.3.1 Basic Linear Search

L'algorithme 1 [Bailey 2005] consiste en une simple recherche linéaire pour trouver un MSS dans un ensemble de clauses insatisfiable. Il commence par partitionner par une affectation la formule initiale en deux sous-ensembles de clauses S et U, où S correspond à l'ensemble de contraintes satisfiable et U correspond à l'ensemble de contraintes insatifiables. Pour chaque contrainte c de U, si l'ensemble $S \cup \{c\}$ est satisfiable, alors la contrainte c est ajoutée à l'ensemble S. Une fois que toutes les contraintes de U sont parcourues, l'ensemble S contiendra un ensemble de clauses maximal satisfiable (MSS). L'algorithme retourne le complément de cet ensemble : le MCS qui lui correspond.

Algorithm 1: Algorithme BLS(Basic Linear Search)

Input : F :An infeasible set of constraints.
Output : A minimal Correction of constraints.
1: $(S, U) \leftarrow$ InitialAssignment(F). % Partitioning by an assignment the original formula into two subsets of clauses S (feasible constraints) and U (infeasible constraints).
2: **for** each $c \in U$ **do**
3: **if** SAT$(S \cup \{c\})$ **then**
4: $S \leftarrow$ S $\cup \{c\}$.
5: **end if**
6: **end for**
7: **return** $F \backslash S$

3.3.2 FastDiag

En suivant le principe diviser pour régner, Felfernig et al. ont proposé un algorithme de diagnostique appelé `FastDiag` [Felfernig 2012]. Le point de différence entre les deux est que `QuickXplain` calcule un IIS/MUS (une explication minimale) et `FastDiag` calcule un MCS (une correction minimale), à partir d'un ensemble de contraintes insatisfaisable.

L'algorithme (voir alg. 2) prend en entrée trois paramètres :
- R : l'ensemble de clauses de référence, il est insatisfaisable.
- T : l'ensemble de clauses où on cherche le MCS afin de satisfaire le reste des contraintes dans R.
- $hasD$: Un booléen pour indiquer si une correction existe.

Cet algorithme peut être appelé avec les ensembles R et T correspondant à toutes les clauses présentes dans la formule en entrée, si on s'intéresse à calculer un MCS. Pour PMaxSAT(Partial Max-SAT) (c'est-à-dire, il y a des clauses dures et moles), cet algorithme est utilisé avec l'ensemble R égal à l'union des contraintes dures et molles, et que l'ensemble T égal aux contraintes moles.

Le but principal de l'algorithme est de minimiser le nombre de tests de faisabilité lors de l'extraction du MCS. Si k est la taille du plus grand MCS et m est la taille de la formule originale, l'algorithme a besoin dans le pire des cas de $(k + k * log(\frac{m}{k}))$ appels au solveur.

Algorithm 2: Algorithme basique de FastDiag (BFD)

Input : R; $T \subseteq R$; $hasD$.
Output : A minimal Correction of constraints.

if $hasD \wedge \text{SAT}(R)$ **then**
  return $\emptyset$
end if
if $|T| = 1$ **then**
  return T
end if
$m \leftarrow |T| \div 2$
$(T_1, T_2) \leftarrow (T_{1..m}, T_{m+1..|T|})$
$D_2 \leftarrow \text{BFD}(R \setminus T_1, T_2, T_1 \neq \emptyset)$
$D_1 \leftarrow \text{BFD}(R \setminus D_1, T_1, D_2 \neq \emptyset)$
return $D_1 \cup D_2$

3.3.3 Approche de Joao Markes-Silva et al.

En pratique, les algorithmes permettant d'énumérer les MCSs peuvent être inefficaces. Pour cela, Joao Mark-Silva et al. [Marques-Silva 2013] ont développé trois techniques permettant d'améliorer les performances des approches existantes (réduire le nombre de tests de faisabilité) :

3.3.3.1 Les techniques proposées par Joao Markes-Silva et al.

1. La recherche d'un ensemble disjoint de cores infaisables.
2. L'exploitation des "backbone litterals".
3. L'exploitation des clauses satisfiables par une affectation.

Exploitation des MUSs disjoints

Proposition 3.3.1 *Soit F un ensemble de clauses insatisfiable, et P une partition de F comme suit, $P = \{G, C_1, ...C_r\}$, où chaque C_i, $i = 1$, ..., r est un infaisable core de F. Donc, r représente une borne inférieure sur la taille de chaque MCS.*

Proposition 3.3.2 *Soit F un ensemble de clauses insatisfiable et une affectation μ. μ permet de partionner F : $\{S, U\}$, où S représente les clauses satisfiables et U les clauses insatisfiables. Ainsi :*
Il existe un MSS W et un MCS M de F tel que $S \subseteq W$ et $U \supseteq M$.

F est insatisfiable et partitionnée en $\{G, C_1, ..., C_r\}$, et $C_1, ..., C_r$ sont des MUSs infaisables. Chaque affectation μ satifaisant G partitionne chaque insatisfiable core C_i en deux ensembles disjoints S_i et U_i, S_i contient les clauses satisfaites et U_i contient les clauses insatisfaites. Nous pourrons maintenant construire l'ensemble $S = G \cup U_i$, pour i = 1, ..., r . Tel que $S \subseteq W$ où W est MSS de F (voir la proposition 3.3.1).

Exploitation des "backbone litterals" Soit M un MCS. Par définition d'un MCS, pour chaque clause $c \in M$, $c = (l_1, l_2, ..., l_k)$, $(F \backslash M) \cup \{c\} \models \bot$. Par conséquent, $F \backslash M \models \neg c$, cela veut dire que pour chaque littéral $l_i \in c$, $F \backslash M \models \neg l_i$. Donc $\neg l_i$ est un backbone littéral [Kilby 2005] pour $F \backslash M$. Nous pourrons exploiter les backbone litterals pour le calcul des MCSs en ajoutant la négation de chaque clause qui peut appartenir à un MCS.

Exploitation des clauses satisfiables par une affectation Cette technique consiste à exploiter les clauses satisfaisables par une affectation satisfaisant un sous ensemble de clauses.

Enhanced Basic FastDiag (EFD) En utilisant les trois nouvelles techniques, Joao Marques-Silva et al. ont proposé l'algorithme 3 qui améliore l'algorithme `FastDiag` [Felfernig 2012]. Cet algorithme utilise un nombre d'ensembles globaux, B contient les clauses représentant les backbone litterals calculés ; S représente le MSS courant calculé ; et les ensembles $U_1, ..., U_r$ représentant les clauses falsifiées pour chaque ensemble de MUSs disjoints. L'ensemble B est mis à jour à chaque fois qu'une clause est ajoutée au MCS (voir

la ligne 7). L'ensemble S est modifié (voir la ligne 4) à chaque fois que le solveur SAT (les ensembles S et B sont ajoutés au solveur) retourne satisfiable.

Algorithm 3: Algorithme Enhanced Basic FastDiag (EFD)

```
1  Input : R ; T ⊆ R ; hasD.
2  Global : S, B, U_1,... ,U_r.
3  Output : A minimal Correction of constraints.
1:  if hasD then
2:    (st, μ) ← SAT(R ∪ S ∪ B)
3:    if st then
4:      UpdateSatisfiedClauses(μ).
5:      return ∅
6:    end if
7:  end if
8:  if |T| = 1 then
9:    UpdateBackboneLiterals(T)
10:   return T
11: end if
12: m ← |T| ÷ 2
13: (T_1, T_2) ← (T_1..m, T_m+1..|T|)
14: D_2 ← BFD(R \ T_1,T_2,T_1 ≠ ∅)
15: D_1 ← BFD(R \ D_1,T_1,D_2 ≠ ∅)
16: return D_1 ∪ D_2
```

3.3.3.2 Le nouvel algorithme de Joao Mark Silva et al.

Joao Mark-Silva et al. [Marques-Silva 2013] ont aussi proposé un nouvel algorithme permettant de calculer un MCS, voir alg. 4. L'idée de cet algorithme vient de l'observation de [Birnbaum 2003] : les clauses falsifiées par une affectation complète ne peuvent pas avoir de littéraux complémentaires.

L'algorithme commence par décomposer l'ensemble de clauses insatisfiable en entrée en deux sous ensemble S et U_i. On crée une clause D qui est la disjonction de toutes les clauses dans U_i. Si le test de faisabilité de $S \cup \{D\}$ est positif, cela signifie qu'il est possible de satisfaire au moins une clause dans U_i quand S est satisfiable. Donc, S ne représente pas un MSS (aussi U_i n'est pas un MCS). Dans ce cas, on enlève les clauses satisfiables de U_i et on les ajoute à S. Dans le cas contraire ($S \cup \{D\}$ est insatisfiable), il n'y a pas de clauses dans U_i qui peuvent satisfaire S. Donc S est un MSS (U_i représente un MCS).
L'exploitation des techniques présentées auparavant peut aussi augmenter les performances de cet algorithme.

L'algorithme assure $m-p+1$ appels au solveur SAT, la taille du plus petit MCS

est p et m est la taille de la formule d'origine.

Algorithm 4: Algorithme pour calculer un MCS avec la clause D (CLD)

Input : i :Target unsatisfiable core.
Global : S, B, U_1, ..., U_r.
Output : A minimal Correction of constraints.

```
st ← true
while st ∧ (U_i > 1) do
    D ← (V_{c∈U_i^c})
    (st, μ) ← SAT(S ∪ B ∪ {D})
    if st then
        UpdateBackboneLiterals(μ)
    end if
end while
UpdateBackboneLiterals(U_i)
return U_i
```

3.3.4 Conclusion

Nous avons présenté deux algorithmes permettant de calculer un seul MCS. Le premier est intuitif, nommé "Basic Linear Search (BLS)", il consiste à appeler de manière itérative le solveur. Le deuxième est `FastDiag` qui se base sur le principe de **Diviser pour régner** avec un nombre logarithmique de tests de faisabilité : la complexité dans le pire des cas de `FastDiag` est $2.log_2(n/d) + 2d$, où d est la taille du MCS trouvé et n est le nombre de contraintes ; la complexité dans le meilleur des cas est $log_2(n/d) + 2d$ [Felfernig 2012].

Les techniques proposées dans [Marques-Silva 2013] peuvent être intégrées dans les algorithmes calculant un MCS pour améliorer ses performances. Nous avons aussi présenté le nouvel algorithme de Joao Mark Silva et al. exploitant les nouvelles techniques proposées. Les résultats rapportés dans [Marques-Silva 2013] ont montré que ces nouveaux algorithmes améliorent de manière significative les algorithme qui existent.

3.4 Algorithmes pour isoler un seul MUS

D'après la définition, un IIS est un ensemble infaisable irréductible. Il dispose de deux propriétés :

- il est lui-même infaisable ;
- tous ses sous-ensembles sont faisables.

Il est irréductible dans le sens où chacune de ses contraintes contribue à l'infaisabilité. Déterminer lesquels des éléments d'un IIS doivent être réparés est une façon de trouver les contraintes erronées dans un système de contraintes infais-

able. Dans cette section, nous décrivons des algorithmes permettant de déterminer un noyau infaisable dans un système infaisable. Nous allons étudier ceux de [Chinneck 2001] [Tamiz 1996] [Guieu 1999] [Junker 2004].

3.4.1 Méthode Deletion Filter

Cet algorithme a été introduit par Chinneck et Dravnieks [Chinneck 1991] (cf. alg. 5). Selon un ordre donné, cet algorithme retire temporairement une contrainte choisie de l'ensemble de contraintes. Si l'ensemble réduit forme un sous-système infaisable, alors il n'y a pas de raison de laisser cette contrainte, elle est donc retirée de façon permanente, sinon elle est réinsérée. On procède ainsi jusqu'à ce qu'on ait parcouru toutes les contraintes.

Deux remarques sont retenues de cet algorithme :
- les seules contraintes retenues dans l'ensemble sont celles dont la suppression rend l'ensemble réalisable ;
- l'efficacité de cet algorithme peut être améliorée si nous réordonnons dynamiquement les contraintes.

Algorithm 5: Algorithme de Deletion Filter

Input : C :An infeasible set of constraints.
Output : A minimal infeasible set of constraints in C.

```
for each constraint c_i in C do
    Temporarily drop the constraint c_i from C.
    Test the feasibility of C\ c_i :
    if feasible then
        return dropped constraint to the set.
    else
        drop the constraint permanently.
    end if
end for
We take the set of constraints that remains or return C.
```

Exemple [Chinneck 2008] Considérons le système de contraintes $\{A, B, C, D, E, F\}$, qui contient le seul IIS $\{B, F, D\}$. Si nous exécutons l'algorithme pour ce système de contraintes, nous obtenons le déroulement décrit dans le tableau de la figure 3.2.

3.4.2 Méthode Additive

`Additive Method` (cf. alg. 6), un algorithme proposé par Tamiz et al [Tamiz 1996]. A l'inverse de `Deletion Filter`, l'algorithme commence avec un ensemble de contraintes I égal à vide.

Itération	La contraint retirée	Le système obtenu	La nature du système obtenu	L'opération effectuée
1	A	$\{\mathbf{B}, C, \mathbf{D}, E, \mathbf{F}, G\}$	Infaisable	A est supprimée définitivement
2	**B**	$\{C, \mathbf{D}, E, \mathbf{F}, G\}$	Faisable	**B** est réinséré
3	C	$\{\mathbf{B}, \mathbf{D}, E, \mathbf{F}, G\}$	Infaisable	C est supprimée définitivement
4	**D**	$\{\mathbf{B}, E, \mathbf{F}, G\}$	Faisable	**D** est réinséré
5	E	$\{\mathbf{B}, \mathbf{D}, \mathbf{F}, G\}$	Infaisable	E est supprimée définitivement
6	**F**	$\{\mathbf{B}, \mathbf{D}, G\}$	Faisable	**F** est réinséré
7	G	$\{\mathbf{B}, \mathbf{D}, \mathbf{F}\}$	Infaisable	G est supprimée définitivement
8	Comme sortie, nous obtenons le IIS $\{\mathbf{B}, \mathbf{F}, \mathbf{D}\}$			

FIGURE 3.2 – Déroulement de `Deletion Filter` pour l'ensemble de contraintes $\{A, B, C, D, E, F\}$ contenant le seul IIS $\{B, F, D\}$.

Selon un ordre donné, on insère des contraintes dans un ensemble T, ce dernier est initialisé par I. Une fois que T devient infaisable, on rajoute la dernière contrainte insérée à I, car T contient au moins un IIS dont la dernière contrainte insérée lui appartient. On procède ainsi jusqu'à ce que I devienne infaisable.

Algorithm 6: Algorithme de Additive Method

Input : C :An infeasible set of constraints.
Output : I : A minimal infeasible set of constraints in C.

1: $T \leftarrow \emptyset$, $I \leftarrow \emptyset$.
2: **while** I feasible **do**
3: $T \leftarrow I$.
4: **for** each constraint c_i in C **do**
5: $T \leftarrow T \cup \{c_i\}$.
6: **if** T infeasible **then**
7: $I \leftarrow I \cup \{c_i\}$.
8: Go to 10.
9: **end if**
10: **end for**
11: **end while**
12: **return** I

Exemple Considérons toujours l'exemple du système de contraintes $\{A, B, C, D, E, F\}$, qui contient le seul IIS $\{B, F, D\}$. Voici l'exécution de l'algorithme Additive Method pour ce système (voir le tableau dans la figure 3.3) :

Itération	La contraint ajoutée	T	La nature du système obtenu	I
1	A	$\{A\}$	Faisable	$\emptyset$
2	**B**	$\{A, \mathbf{B}\}$	Faisable	$\emptyset$
3	C	$\{A, \mathbf{B}, C\}$	Faisable	$\emptyset$
4	**D**	$\{A, \mathbf{B}, C, \mathbf{D}\}$	Faisable	$\emptyset$
5	E	$\{A, \mathbf{B}, C, \mathbf{D}, E\}$	Faisable	$\emptyset$
6	**F**	$\{A, \mathbf{B}, C, \mathbf{D}, E, \mathbf{F}\}$	Infaisable	$\{\mathbf{F}\}$
7	A	$\{\mathbf{F}, A\}$	Faisable	$\{\mathbf{F}\}$
8	**B**	$\{F, A, \mathbf{B}\}$	Faisable	$\{\mathbf{F}\}$
9	C	$\{\mathbf{F}, A, \mathbf{B}, C\}$	Faisable	$\{\mathbf{F}\}$
10	**D**	$\{\mathbf{F}, A, \mathbf{B}, C, \mathbf{D}\}$	Infaisable	$\{\mathbf{F}, \mathbf{D}\}$
11	A	$\{\mathbf{F}, \mathbf{D}, A\}$	Faisable	$\{\mathbf{F}, \mathbf{D}\}$
12	**B**	$\{\mathbf{F}, \mathbf{D}, A, \mathbf{B}\}$	Infaisable	$\{\mathbf{F}, \mathbf{D}, \mathbf{B}\}$
13	La sortie représente le IIS $\{\mathbf{B}, \mathbf{F}, \mathbf{D}\}$			

* T : l'ensemble de test courant de contraintes.
* I : l'ensemble des membres IIS identifiés jusqu'ici.

FIGURE 3.3 – Déroulement de `Additive Method` pour l'ensemble de contraintes $\{A, B, C, D, E, F\}$ contenant le seul IIS $\{B, F, D\}$.

3.4.3 Méthode hybride Additive et Deletion Filter

En combinant les deux algorithmes précédents, nous obtenons ce troisième algorithme d'isolation d'infaisabilité (cf. alg. 7), il est proposé par Guieu et Chinneck [Guieu 1999]). Il suffit de lancer la méthode additive jusqu'à ce que l'infaisabilité soit détectée et puis changer pour `Deletion Filter` pour trouver l'IIS.

Exemple La tableau dans la figure 3.4 déroule l'algorithme qui combine les deux méthodes : Additive method et Deletion Filter. Nous utilisons le système de contrainte $\{A, B, C, D, E, F\}$, qui contient le seul IIS $\{B, F, D\}$.

3.4.4 Discussion

Ces trois algorithmes sont parmi les premiers travaux dans la communauté de recherche opérationnelle qui ont été proposés pour calculer un IIS/MUS. Ce sont des algorithmes itératifs. On trouve aussi dans la littérature `Elastic Filter` [Chinneck 2008], un algorithme qui utilise systématiquement des variables d'écart pour identifier dans la première phase du Simplexe les contraintes susceptibles de figurer dans un IIS.
Junker [Junker 2004] a proposé un algorithme générique basé sur une stratégie "Divide-and-Conquer" pour calculer efficacement un IIS/MUS lorsque la taille des sous-ensembles conflictuels est beaucoup plus petite que celle de l'ensemble total des contraintes.

Algorithm 7: Algorithme qui combine Additive Method et Deletion Filter

Input : C :An infeasible set of constraints.
Output : T : A minimal infeasible set of constraints in C.

Set $T \leftarrow \emptyset$.
while (T feasible **and** $C \neq \emptyset$) **do**
  Let $c_i \in C$.
  Set $T \leftarrow T \cup c_i$.
  Set $C \leftarrow C \setminus c_i$.
end while
for each constraint t_i in $t_{|T|-1}$ in T : **do**
  Temporarily drop the constraint t_i.
  Test the feasibility of $T \setminus t_i$:
  if feasible **then**
    return dropped constraint to T.
  else
    $T \leftarrow T \setminus t_i$.
  end if
end for
return T

Étape 1 : `Additive Method` pour le système $\{\mathbf{A}, \mathbf{B}, C, \mathbf{D}, E, F\}$				
Itération	La contraint ajoutée	T	La nature du système obtenu	I
1	**A**	$\{\mathbf{A}\}$	Faisable	$\emptyset$
2	**B**	$\{A, \mathbf{B}\}$	Faisable	$\emptyset$
3	C	$\{\mathbf{A}, \mathbf{B}, C\}$	Faisable	$\emptyset$
4	**D**	$\{\mathbf{A}, \mathbf{B}, C, \mathbf{D}\}$	Infaisable	$\{\mathbf{D}\}$
Étape 2 : `Deletion Filter` pour le système $\{\mathbf{A}, \mathbf{B}, C, \mathbf{D}\}$				
Itération	La contraint retirée	Le système obtenu	La nature du système obtenu	L'opération effectuée
5	**A**	$\{\mathbf{B}, C, \mathbf{D}\}$	Faisable	**A** est réinsérée
6	**B**	$\{\mathbf{A}, C, \mathbf{D}\}$	Faisable	**B** est réinséré
7	C	$\{\mathbf{A}, \mathbf{B}, \mathbf{D}\}$	Infaisable	C est supprimée définitivement
8	**D**	$\{\mathbf{A}, \mathbf{B}\}$	Faisable	**D** est réinséré
9	Comme sortie, nous obtenons le IIS $\{\mathbf{A}, \mathbf{B}, \mathbf{D}\}$			

FIGURE 3.4 – Déroulement de `Additive-Deletion Filter` pour l'ensemble de contraintes $\{A, B, C, D, E, F\}$ contenant le seul IIS $\{A, B, D\}$.

3.4.5 Algorithme QUICKXPLAIN

En se basant sur le principe de diviser pour régner, Ulrich Junker [Junker 2004] a proposé une méthode permettant de trouver un ensemble minimal conflictuel de contraintes dans un ensemble de contraintes infaisable, appelée `QUICKXPLAIN` (cf. alg. 8). `QUICKXPLAIN` accélère considérablement les méthodes de base et assure une bonne scalabilité. Dans le cas où les IISs sont de petites tailles par rapport au nombre de contraintes, `QUICKXPLAIN` permet de réduire le nombre de tests de faisabilité effectués. Cette méthode tient compte des préférences de l'utilisateur sur les contraintes, on écrit $c_i \prec c_j$ ssi la contrainte c_i est préférée (plus importante) que la contrainte c_j.

L'algorithme prend trois éléments en paramètre :
- B : contient les contraintes dites dures.
- C : contient les contraintes considérées comme moles.
- $\prec$: une relation d'ordre de préférence entre les contraintes.

Il permet de retourner un conflit préféré de (B,C,$\prec$). L'idée principale de l'algorithme est de minimiser le nombre de tests de faisabilité, et de fournir en sortie un sous ensemble minimal de contraintes en contradiction avec B tout en respectant $\prec$.

Par un processus dichotomique, l'algorithme subdivise le problème à chaque fois en deux sous-problèmes. Il utilise une fonction *split* pour diviser $C =< \alpha_1, ..., \alpha_n >$ en $C_1 =< \alpha_1, ..., \alpha_k >$ et $C_2 =< \alpha_{k+1}, ..., \alpha_n >$, où $1 \leq k < n$. La propriété suivante explique l'assemblage de la sortie des deux sous-problèmes [Junker 2004] :

Proposition 3.4.1 *Soit C_1 et C_2 deux sous-ensembles disjoints de C tels qu'il n'y pas de contraintes dans C_2 qui est plus importante qu'une contrainte dans C_1.*

1. *Si Δ_2 est un conflit préféré de ($B \cup C_1$,C_2,$\prec$) et Δ_1 est un conflit préféré de ($B \cup \Delta_2$,C_1,$\prec$), alors $\Delta_1 \cup \Delta_2$ est un conflit préféré de (B,C,$\prec$).*

Remarque Dans l'approche de Junker, les contraintes appartenant à l'ensemble B sont considérées comme dures. Donc, lorsque B est vide et C est inconsistant, la sortie de `QUICKXPLAIN` est un IIS.

Exemple Considérons à nouveau l'ensemble des contraintes $\{A, B, C, D, E, F\}$ où $\{B, F, D\}$ est un MUS. Cet ensemble ne dispose pas de contraintes obligatoires et $\prec$ représente un ordre lexicographique. L'exécution de l'algorithme `QUICKXPLAIN` pour cet exemple est déroulée dans la figure 3.5.

Algorithm 8: Algorithme de QUICKXPLAIN

Input : B : hard constraints ; C : soft constraints ; $\prec$: An order relationship between constraints.

Output : A preferred conflict of (B,C,$\prec$).

QUICKXPLAIN(B,C,$\prec$)

1: **if** isConsistent($B \cup C$) **return** 'no conflict' ;
2: **else if** $C = \emptyset$ **then return** $\emptyset$;
3: **else return** QUICKXPLAIN'(B,B,C,$\prec$) ;

QUICKXPLAIN'(B,Δ,C,$\prec$)

1: **if** $\Delta \neq \emptyset$ and not isConsistent(B) **then return** $\emptyset$;
2: **if** $C = \{\alpha\}$ **then return** α ;
3: let $\alpha_1, ..., \alpha_n$ be an enumeration of C that respects $\prec$;
4: let k be $split(n)$ where $1 \leq k < n$.
5: $C_1 \leftarrow \{\alpha_1, ..., \alpha_k\}$ and $C_2 \leftarrow \{\alpha_{k+1}, ..., \alpha_n\}$.
6: $\Delta_2 \leftarrow$ QUICKXPLAIN'($B \cup C_1$,C_1,C_2,$\prec$)
7: $\Delta_1 \leftarrow$ QUICKXPLAIN'($B \cup \Delta_2$,Δ_2,C_1,$\prec$)
8: **return** $\Delta_1 \cup \Delta_2$

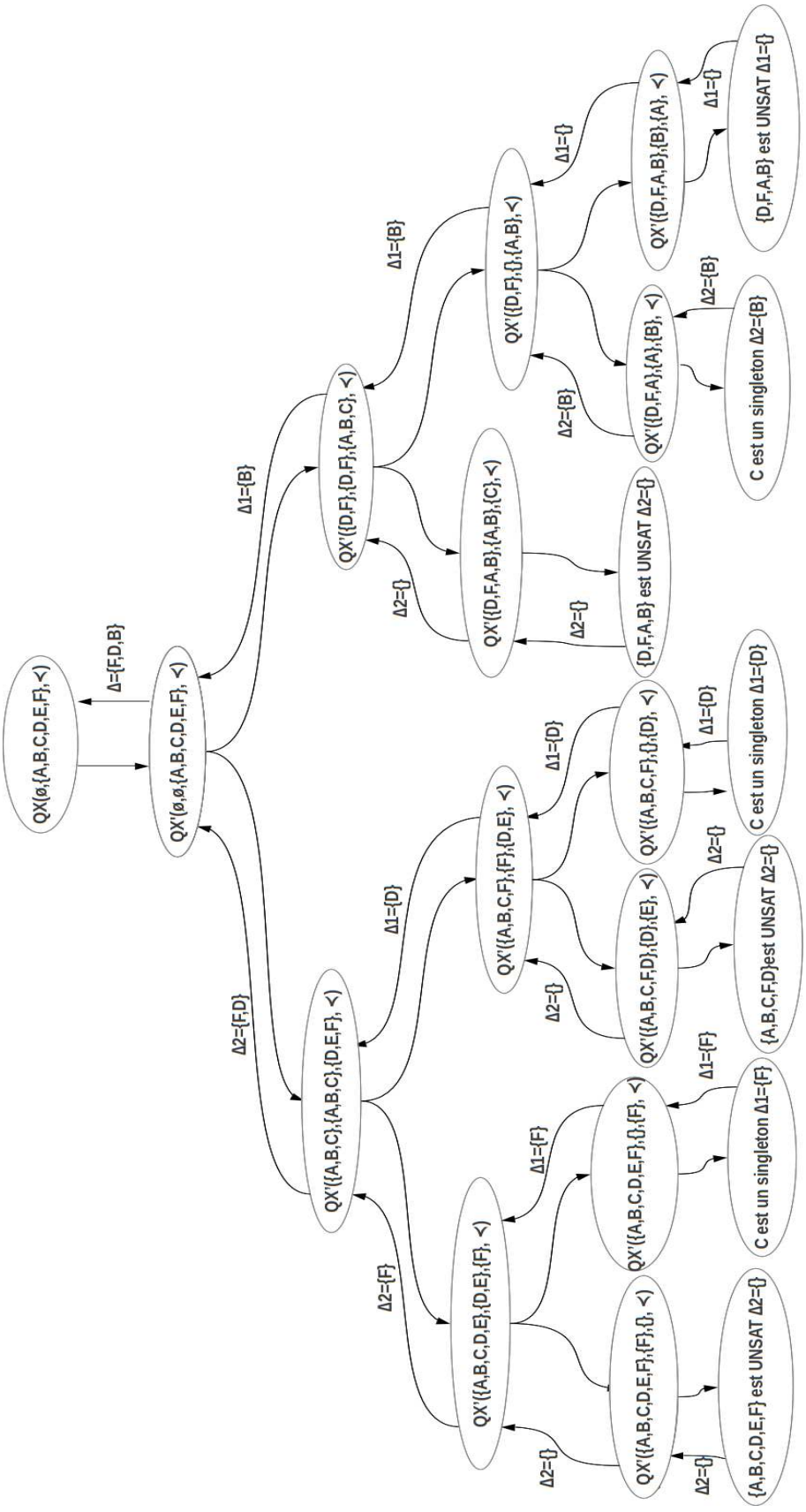


FIGURE 3.5 – La trace d'exécution de `QUICKXPLAIN` (voir alg. 8) pour $B = \emptyset$ et $C = \{A, B, C, D, E, F\}$ où $\{B, F, D\}$ est un IIS/MUS. QX et QX' représentent respectivement les fonctions QUICKXPLAIN et QUICKXPLAIN' dans l'algorithme.

3.4.6 Conclusion

Les quatre algorithmes se basent sur le principe du test de la faisabilité d'un sous-système de contraintes. Ainsi, ils sont considérés comme des méthodes générales logiques d'isolation d'IISs qui ne dépendent d'aucune propriété du système de contraintes. Avec un test de faisabilité correct et complet, les quatre algorithmes permettent de retourner exactement un seul IIS pour un système de contraintes infaisable. La différence réside dans le nombre de tests de faisabilité effectué. Supposant que la cardinalité de l'ensemble de contraintes en entrée est n, et la cardinalité de l'ensemble retourné est k, nous présentons dans le tableau 3.6 le nombre de tests de faisabilité de chaque méthode, dans le meilleur cas et dans le pire cas. Les informations à propos du nombre de tests de faisabilité de `QUICKXPLAIN` sont tirées du papier de Ulrich Junker [Junker 2004].

La méthode		Le meilleur cas	Le pire cas
`Deletion Filter`		n	n
`Additive Method`		$\sum_{i=1}^{k} i + 1$	$\sum_{i=0}^{k-1}(n-i)+1$
`Additive/Deletion Method`		$k + (k-1)$	$n + (n-1)$
`QUICKXPLAIN`	$split(n) = n/2$ $split(n) = n-1$ $split(n) = 1$	$log(n/k) + 2k$ $2k$ k	$2k * log(n/k) + 2k$ $2n$ $n + k$

FIGURE 3.6 – Le nombre de tests de faisabilité pour `Deletion Filter`, `Additive Method`, `Additive/Deletion Method` et `QUICKXPLAIN`.

3.5 Algorithmes pour trouver plusieurs MCSs

Les méthodes présentées dans la section 3.3 permettent de détecter un seul MCS. Pour corriger les erreurs dans un système de contraintes inconsistant, un seul MCS peut être insuffisant. Pour cela, nous explorons dans cette section les techniques les plus utilisées pour trouver plusieurs MCSs.

L'énumération des MCSs peut être mise en oeuvre par plusieurs appels à une méthode calculant un seul MCS, en bloquant chaque MCS calculé avec une clause dure qui désactive la répétition de ce même MCS dans les prochains appels [Marques-Silva 2013].

L'algorithme DAA(voir alg. 10), qui exploite la relation entre MCSs et MUSs comme CAMUS, permet aussi d'énumérer les MCSs (compléments des MSSs) d'un système de contraintes ; il les calcule à l'aide de la méthode **grow**. La différence est que CAMUS procède en deux étapes séparées : il calcule d'abord les MCSs, après les MUSs. Alors que DAA calcule les deux ensembles tout au long de son exécution. Comme DAA et CAMUS, l'algorithme MARCO (voir alg. 14) énumère également tous les MSSs. Dans la section suivante, nous décrivons la première phase de l'algorithme CAMUS.

3.5.1 Approche de M.H.Liffiton

CAMUS [Liffiton 2008] commence par calculer tous les MCSs en résolvant successivement un problème d'optimisation similaire à celui du MaxSAT, par l'utilisation d'un solveur SAT incrémental et des informations à propos de l'apprentissage de clauses, voir l'algorithme 9. L'algorithme résout successivement un problème similaire à celui du MaxSAT pour calculer les MCSs. A chaque itération, il cherche un ensemble maximal de clauses faisable (MSS), le complément de cet ensemble est un MCS. Il ajoute ensuite les contraintes permettant d'exclure les résultats obtenus précédemment, jusqu'à ce qu'il ne reste aucun sous-ensemble de clauses faisable.

L'instruction à la ligne 1 de l'algorithme consiste à augmenter la formule CNF en entrée ϕ avec des variables dites **clause-selector**, pour que les solveurs SAT standards puissent la manipuler, la formule construite est ϕ'. Chaque clause C_i dans la formule CNF est augmentée avec la négation d'une variable **clause-selector** y_i pour donner $C'_i = \neg y_i \vee C_i$ dans la nouvelle formule ϕ'. Affecter une variable y_i avec la valeur TRUE implique la clause originale. En revanche, Affecter à y_i avec la valeur FALSE implique la suppression de la clause. Chaque itération de la boucle **while** (lignes 4-12) permet de trouver tous les MCSs de taille k, k est incrémenté de 1 après chaque itération. En effet, un MCS est obtenu en cherchant une affectation satisfaisant ϕ' avec un ensemble minimal de variables **clause-selector** y_i affectées avec FALSE. La contrainte AtMost($\{\neg y_1, \neg y_2, ..., \neg y_n\}$,$k$) est ajoutée à la formule ϕ', la formule créée est ϕ'_k, l'objectif est de limiter le nombre de variables **clause-selector** pouvant être assignées à FALSE, voir la ligne 5. La boucle WHILE (lignes 6-10) cherche exhaustivement tous les MCSs de taille k.

Algorithm 9: Algorithme pour trouver tous les MCSs d'une formule ϕ

Input : ϕ : An infeasible SAT formula in CNF form.
Output : $MCSs$: The set that contain all MCSs in ϕ.

1: $\phi' \leftarrow$ AddYVars(ϕ).
2: $MCSs \leftarrow \emptyset$.
3: $k \leftarrow 1$.
4: **while** SAT(ϕ') **do**
5: $\quad \phi'_k \leftarrow \phi' \wedge$ AtMost($\{\neg y_1, \neg y_2, ..., \neg y_n\}$,$k$).
6: $\quad$ **while** $newMCS \leftarrow$ IncrementalSAT(ϕ'_k) **do**
7: $\quad\quad MCSs \leftarrow MCSs \cup \{newMCS\}$.
8: $\quad\quad \phi'_k \leftarrow \phi'_k \wedge$ BlockingClause(newMCS).
9: $\quad\quad \phi' \leftarrow \phi' \wedge$ BlockingClause(newMCS).
10: $\quad$ **end while**
11: $\quad k \leftarrow k + 1$.
12: **end while**
13: **return** $MCSs$

Après chaque MCS trouvé (ligne 7), une clause de blocage est ajoutée à ϕ' et ϕ'_k, pour empêcher de trouver cette dernière dans les prochaines itérations. La condition

de la boucle WHILE (ligne 4) vérifie que la formule ϕ' augmentée avec toutes les clauses de blocage est restée faisable, avec aucune limite sur le nombre de variables y_i.

Exemple [Liffiton 2008] Soit un système de contrainte $C = \{C_1, C_2, C_3, C_4, C_5, C_6\}$. C contient cinq MCSs : $\{C_1\}$, $\{C_2, C_3, C_5\}$, $\{C_2, C_3, C_6\}$, $\{C_2, C_4, C_5\}$, $\{C_2, C_4, C_6\}$. Exécutons maintenant l'algorithme de Liffiton et Sakallah (voir alg. 9) pour isoler les MCSs dans C :

- Dans la première itération, l'algorithme cherche les MCSs de taille 1 en ajoutant la contrainte AtMost avec $k = 1$. Le seul MCS de taille 1 qui sera isolé de C est $\{C_1\}$ qui est la solution MaxSAT. Aucun autre MCS de taille 1 ne peut être trouvé après l'ajout de la clause de blocage qui correspond à $\{C_1\}$. La deuxième itération ne permet de trouver aucun nouveau MCS. Après, l'algorithme sort de la boucle. La borne k est incrémentée à 2.
- Dans la deuxième itération, aucun MCS de plus n'est trouvé ; en d'autres termes, il n'y a aucun MCS de taille 2 dans C.
- Dans la troisième itération, l'algorithme trouve le reste des MCSs : $\{C_2, C_3, C_5\}$, $\{C_2, C_3, C_6\}$, $\{C_2, C_4, C_5\}$, $\{C_2, C_4, C_6\}$. Une fois tous les MCSs trouvés, l'algorithme sort de la boucle **While** principale car ϕ' avec toutes les clauses de blocage devient infaisable.

3.5.2 Conclusion

L'énumération des MCSs trouvent de nombreuses applications. Par exemple, un MCS avec le plus petit nombre de contraintes représente une solution du problème de Satisfiability Maximale (MaxSAT). Un autre usage des MCSs est l'énumération des MUSs en exploitant la dualité qui existe entre ces deux ensembles. La plupart des approches qui existent utilisent ou bien un solveur MaxSAT, ou bien des appels itératifs à un solveur normal (SAT ou de contraintes) [Marques-Silva 2013]. CAMUS qui se base sur MaxSAT est parmi les approches les plus efficaces qui ont été proposées pour trouver tous les MCSs. Elle résout une série de problèmes de MaxSAT consécutifs, chacun avec la restriction qui exclut les MCSs trouvés précédemment, jusqu'à ce qu'il ne reste plus d'ensembles satisfiables. Elle utilise également un solveur incrémental pour réduire le coût de la résolution.

3.6 Algorithmes pour trouver de multiples MUSs

Les algorithmes 5, 6, 7, 8 permettent de retourner exactement un seul IIS. Cependant, le système de contraintes C peut avoir plusieurs infaisabilités, et donc plusieurs IISs, alors il est très important de les isoler.

Une méthode simple pour trouver plusieurs IISs consiste à procéder de manière cyclique [Chinneck 2008] :

1. isoler un IIS ;
2. déterminer une correction pour ce IIS ;
3. si le système est toujours infaisable, aller à l'étape (1).

En changeant l'ordre des contraintes dans C, les algorithmes présentés dans la section 3.4 peuvent nous retourner d'autres IISs. Une méthode naïve pour obtenir tous les IISs consiste à essayer tous les réarrangements (permutations) possibles. Le nombre de choix possibles de permutations dans un certain ordre est $n!$, où $n = |C|$.

Parmi les premiers travaux, pour trouver tous les IIS/MUSs, nous en citons certain dans le domaine du diagnostique utilisant l'arbre $CS - tree$ ceux de Hou [Hou 1994] et de Han et Lee [Han 1999]. Ces deux méthodes énumèrent explicitement et testent la faisabilité de chaque sous-ensemble du système de contrainte insatisfiable. La racine de l'arbre $CS - tree$ représente C, et chaque nœud n représente un sous-ensemble S de C. Pour chaque contrainte c dans S, n a un nœud fils qui représente l'ensemble $S \backslash c$. S'il y a un nœud n représentant un sous-ensemble S de C, et que S est infaisable, et chaque nœud fils de n représente un ensemble de contraintes faisable, alors S est un IIS/MUS.

La méthode de Han et Lee est une amélioration de celle de Hou [Hou 1994]. La méthode de Hou dépend de l'ordre de la génération dans l'arbre $CS - tree$, et il y a des cas où elle peut rater des IISs (méthode incomplète). Les deux méthodes consiste à générer l'arbre $CS - tree$ de C, tout en appliquant un certain nombre de règles. Le rôle d'une règle est d'arrêter l'exploration d'un ou de plusieurs nœuds, sur une certaine condition. Dans la description des procédures de Hou, et de Han et Lee, "**fermer** un nœud" prend le sens d' "arrêter l'exploration du nœud".

3.6.1 Algorithme DAA

L'algorithme DAA (the Dualize and Advance Algorithm) de Bailey et Stuckey [Bailey 2005] (cf. alg. 10) exploite la dualité de couverture par ensembles entre les MCSs et MUSs. DAA utilise la méthode **grow**, équivalente à Basic Linear Search (cf. alg. 1), pour trouver les MSSs et leurs MCSs complémentaires. Il calcule ensuite les ensembles de couvertures minimales des MCSs trouvés jusqu'à présent. Ce problème est équivalent au problème qui consiste à calculer toutes les couvertures minimales (minimal transversals) d'un hypergraphe. Cette méthode prend comme paramètre deux ensembles :

– Un ensemble de contraintes insatisfiable C.
– Un ensemble de contraintes satisfiable $seed \subseteq C$.

Après chaque MCS trouvé, l'algorithme DAA calcule la couverture minimale des MCSs obtenus en utilisant la méthode **grow**. Puis, il teste la faisabilité de chaque couverture. Même si l'ensemble des MCSs est incomplet, une couverture minimale insatisfaisable représente un MUS. Si au moins un sous-ensemble satisfiable est trouvé parmi les couvertures minimales calculées, alors il sera considéré comme le prochain point de départ pour la méthode **grow** afin de trouver un autre MSS/MCS.

De cette façon, l'approche de Bailey et Stuckey [Bailey 2005] calcule les MCSs et les MUSs tout au long de l'exécution de leur algorithme.

Algorithm 10: Algorithme DAA pour énumérer les MSSs et MUSs d'un ensemble de contraintes

Input : C :unsatisfiable constraint set.
Output : MSSs and MUSs of C as they are discovered.

$MCSs$,$MUSs$,$seed \leftarrow \emptyset$
$haveSeed \leftarrow True$
while haveSeed **do**
  $MSSs \leftarrow$ grow($seed$,C)
  yield $MSSs$
  $MCSs \leftarrow \cup \{C \backslash MSS\}$ % the complement of an MSS is an MCS
  $haveSeed \leftarrow False$
  for $candidate \in$ (HittingSets($MCSs$) $\setminus$ $MUSs$) **do**
    if candidate is satisfiable **then**
      $seed \leftarrow candidate$
      $haveSeed \leftarrow True$
    else
      yield $candidate$
      $MUSs \leftarrow MUSs \cup \{candidate\}$
    end if
  end for
end while
return $MUSs$, $MSSs$

Exemple [Liffiton 2015] Nous allons, à travers cet exemple, exécuter l'algorithme DAA sur le système de contraintes suivant : $C = \{C_1 : (a), C_2 : (\neg a), C_3 : (\neg a \vee b), C_4 : (\neg b)\}$. Le déroulement est comme suit :

- **grow**$(\emptyset) \rightarrow$ MSS : $\{C_1, C_3\}$, le MCS correspondant : $\{C_2, C_4\}$
- MCSs = $\{\{C_2, C_4\}\}$
- HittingSets($MCSs$) = $\{\{C_2\}, \{C_4\}\}$

Les couvertures calculées $\{\{c_2\}, \{C_4\}\}$ sont satisfiables. Un de ces deux ensembles (exemple $\{C_2\}$) est considéré comme le point de départ suivant pour la méthode **grow**.

- **grow**$(\{C_2\}) \rightarrow$ MSS : $\{C_2, C_3, C_4\}$, le MCS correspondant : $\{C_1\}$
- MCSs = $\{\{C_2, C_4\}, \{C1\}\}$
- HittingSets($MCSs$) = $\{\{C_1, C_2\}, \{C_1, C_4\}\}$

La première couverture $\{C_1, C_2\}$ est insatisfaisable, l'autre ($\{C_1, C_4\}$) est satisfaisable. Donc, $\{C_1, C_2\}$ est affiché comme un MUS. $\{C_1, C_4\}$ est utilisé comme un autre point de départ pour **grow**.

- **grow**$(\{C_1, C_4\}) \rightarrow$ MSS : $\{C_1, C_4\}$, le MCS correspondant : $\{C_2, C_3\}$

- MCSs = $\{\{C_2, C_4\}, \{C1\}, \{C_2, C_4\}\}$
- HittingSets($MCSs$) = $\{\{C_1, C_2\}, \{C_1, C_3, C_4\}\}$

Les couvertures minimales trouvées sont toutes insatisfaisables (la première représente un MUS déjà trouvé). Ainsi, $\{C_1, C_3, C_4\}\}$ est affiché comme un nouveau MUS trouvé. Puisque l'algorithme a trouvé tous les MUSs, donc il s'arrête.

3.6.2 Algorithme CAMUS

M.H.Liffiton [Liffiton 2005, Liffiton 2008, Liffiton 2009] propose une approche correcte et complète permettant d'énumérer tous les MUSs inclus dans un système de contraintes, appelée CAMUS (Compute All Minimal Unsatisfiable Subsets). CAMUS exploite la relation entre un ensemble faisable et infaisable de contraintes qui peut être utilisée pour l'extraction des MUSs [De Kleer 1987, Reiter 1987].

Pour générer tous les MUSs, CAMUS calcule d'abord l'ensemble complet de tous les MCSs, et il calcule toutes les couvertures irréductibles de cet ensemble. La seconde étape de l'approche CAMUS consiste à produire tous les MUSs en cherchant tous les ensembles irréductibles couvrant tous les MCSs obtenus dans la première étape.

Calculer un seul MUS [Liffiton 2008] Pour calculer un seul MUS, il faut calculer un ensemble de clauses qui couvre tous les MCSs et que cet ensemble trouvé soit irréductible(minimal). Chaque clause dans le MUS obtenu doit être la seule représentante d'au moins un MCS. Prenons l'exemple $\{\{C_1, C_2, C_3\}, \{C_2, C_4\}\}$, une sélection aléatoire d'au moins une contrainte de chaque sous-ensemble génère une couverture qui peut ne pas être minimale, exemple $\{C_1, C_2\}$ (C_1 est redondant). Pour cela, l'approche de M.H.Liffiton force chaque clause sélectionnée à ne pas être redondante. Exemple, soit un ensemble de MCSs $\{\{C_1, C_2, C_3\}, \{C_2, C_4\}, \{C_2, C_5\}\}$, on sélectionne la clause C_3 pour qu'elle soit dans le MUS. Elle appartient seulement au premier MCS, donc on force C_3 pour qu'elle soit non-redondante par la suppression de C_1 et C_2. Cette étape garde $\{\{C_4\}, \{C_5\}\}$ comme sous-problème restant. L'algorithme 11 (nommé PropagateChoice) propage le choix d'une clause et du MCS qui la contient. L'algorithme 12 calcule un seul IIS en se basant sur l'algorithme

cité précédemment.

Algorithm 11: Algorithme pour modifier des MCSs et faire le choix de la clause *thisClause* irredondante en tant que seul élément couvrant *thisMCS*

Input : $MCSs$: The set that contain all MCSs in ϕ ; *thisClause* : A clause in MCSs ; *thisMCS* : The MCS in which *thisClause* apears.

1: **for each** $clause \in thisMCS$ **do**
2: **for each** $testMCS \in MCSs$ **do**
3: **if** $clause \in testMCS$ **then**
4: $testMCS \leftarrow testMCS$ - $\{clause\}$.
5: **end if**
6: **end for**
7: **end for**
8: **for each** $testMCS \in MCSs$ **do**
9: **if** $thisClause \in testMCS$ **then**
10: $MCSs \leftarrow MCSs$ - $\{testMCS\}$.
11: **end if**
12: **end for**
13: MaintainNoSupersets($MCSs$)

Algorithm 12: Algorithme retournant un seul MUS dans un ensemble de MCSs

Input : $MCSs$: The set that contain all MCSs in ϕ.
Output : MUS : A MUS from a set of MCSs.

1: $MUS \leftarrow \emptyset$.
2: **while** $MCSs \neq \emptyset$ **do**
3: $selClause \leftarrow$ SelectRemainingClause($MCSs$). % Selecting arbitrarily a clause in $MCSs$.
4: $selMCS \leftarrow$ SelectMCSContaining($MCSs$,$selClause$). % Select arbitrarily a MCS set in which *selClause* belongs.
5: $MUS \leftarrow MUS \cup \{selClause\}$.
6: PropagateChoice(MCSs,selClause,selMCS)
7: **end while**
8: **return** MUS.

Calculer tous les MUSs À partir de l'algorithme de recherche d'un seul MUS, M.H.Liffiton a fondé l'algorithme permettant de calculer tous les MUSs récursivement, notant que la sélection d'une clause et d'un MCS dans lequel elle appartient est arbitraire, voir la ligne 3 et 4 dans l'algorithme 12. L'algorithme 13 (nommé All-MUSs) prend en entrée l'ensemble des MCSs restants et le MUS couramment calculé dans n'importe quelle branche de la récursion (initialisés à la racine de la récursion par l'ensemble de tous les MCSs et l'ensemble vide, respectivement). Quand il ne reste pas d'éléments dans l'ensemble $MCSs$ (le critère d'arrêt, voir les lignes 1-4), il

affiche le MUS construit dans la branche de la récursivité en cours et retourne pour explorer d'autres branches. Les lignes 5-12 itèrent sur tous les choix possibles d'une clause qui est ajoutée au nouveau MUS et un MCS dans le quel elle appartient. La fonction PROPAGATECHOICE est appelée pour une copie du MCSs courant.

Algorithm 13: Algorithme pour calculer l'ensemble complet des MUSs dans un ensemble de MCSs

1 **Input :** $MCSs$: The set that contain all MCSs in ϕ ; $currentMUS$: The MUS currently being constructed in each branch of the recursion.
2 **Output :** $MUSs$: The complete set of MUSs from a set of MCSs.

```
1:  if MRSs == ∅ then
2:    print currentMUS.
3:    return
4:  end if
5:  for each selClause ∈ RemainingClause(MCSs) do
6:    newMUS ← currentMUS ∪ selClause.
7:    for selMCS ∈ MCSs such that selClause ∈ selMCS do
8:      newMCSs ← MCSs.
9:      PropagateChoice(newMCSs,selClause,selMCS).
10:     AllMUSs(newMCSs,newMUS)
11:   end for
12: end for
13: return
```

3.6.3 Algorithme MARCO

Nous allons expliquer l'approche MARCO [Liffiton 2013] permettant une énumération rapide des MUSs. L'algorithme présenté est une amélioration des travaux antérieurs basés sur l'énumération des sous-ensembles (Hou [Hou 1994], Banda et al. [de la Banda 2003]), DAA [Bailey 2005], CAMUS [Liffiton 2008].

Étant donné un ensemble de contraintes insatisfaisable C, l'idée de l'algorithme consiste à explorer efficacement l'ensemble des parties de C ($P(C)$), de telle sorte à considérer chaque sous partie comme une Algèbre de Boole. Il utilise une fonction $f : P(C) \rightarrow \{0, 1\}$ représentée par une formule propositionnelle de $|C|$ variables, ayant une variable propositionnelle x_i pour chaque contrainte $C_i \in C$. Pour un ensemble $C' \subseteq C$, $f(C') = 1$ ssi l'ensemble C' n'est pas encore exploré. Dans l'algorithme MARCO (cf. alg. 14), la fonction f est stockée comme une formule propositionnelle (Map), et peut être vue comme une carte de $P(C)$, d'où le nom MARCO (**M**apping **R**egions of **Co**nstraint sets) de l'algorithme. La formule Map est initialement toujours vraie (une tautologie), cela signifie que tous les sous ensembles de C n'ont pas encore été explorés. Chaque modèle de Map représente un sous-ensemble de $P(C)$ non exploré et dont la faisabilité est inconnue. MARCO projette une affectation satisfaisant Map sur C pour trouver un sous-ensemble non exploré,

noté *seed* (ligne 4). Si *seed* est satisfaisable, alors il devra être un sous-ensemble d'un MSS, qui peut être identifié en utilisant la méthode **grow** (ligne 6) ; sinon il devra contenir un MUS (IIS), qui peut être identifié en utilisant la méthode **shrink** (ligne 11). Après chaque MSS ou MUS obtenu, cet algorithme ajoute une clause à la formule Map indiquant qu'une région a été explorée (ligne 8 et 12). Soit un MSS M trouvé, afin d'obtenir un autre MSS, MARCO ajoute une information indiquant que le nouveau MSS devra contenir au moins une contrainte qui n'appartient pas à M. Pour cela, l'algorithme utilise la formule suivante : blockDown(M) $\equiv \bigvee_{i:C_i \notin M} x_i$. Pour un MUS W, on ajoute une clause de blocage pour bloquer tous les sur-ensembles de W. Pour cela, l'algorithme utilise la formule suivante : blockUp(W) $\equiv \bigvee_{i:C_i \in W} \neg x_i$.

Pour résumer, voici les étapes de chaque itération de l'algorithme MARCO [Liffiton 2015] :

- Sélectionner un sous-ensemble inexploré dans $P(C)$, un sous-ensemble de C noté *seed* ;
- Tester la satisfiabilité de *seed* ;
- Augmenter ou diminuer l'ensemble *seed* à un MSS ou à un MUS s'il est respectivement satisfaisable ou insatisfaisable ;
- Marquer la région correspondante dans Map comme explorée.

Chaque itération permet d'identifier un nouveau MUS ou MSS. L'algorithme se termine lorsque tous les sous-ensembles ont été explorés et tous les MUSs et MSSs ont été trouvés.

Algorithm 14: Algorithme MARCO pour énumérer les MSSs et les MUSs d'un ensemble de contraintes

1 **Input :** $C = \{C_1, ..., C_n\}$:unsatisfiable constraint set.
2 **Output :** MSSs and MUSs of C.

1: $Map \leftarrow$ BoolFormula($nvars = |C|$) % Empty formula over $|C|$ Boolean variables
2: **while** Map is satisfiable **do**
3: $m \leftarrow$ getModel(Map)
4: $seed \leftarrow \{C_i \in C : m[x_i] = True\}$
5: **if** seed is satisfiable **then**
6: $MSS \leftarrow$ grow(seed,C)
7: **yield** MSS
8: $Map \leftarrow Map \wedge$ blockDown(MSS)
9: **else**
10: $MUS \leftarrow$ shrink(seed,C)
11: **yield** MUS
12: $Map \leftarrow Map \wedge$ blockUp(MUS)
13: **end if**
14: **end while**

Initialisation	$Map \leftarrow$ensemble vide sur $\{x_1, x_2, x_3, x_4\}$
Itération 1	$Map = \top$: SAT getModel $\rightarrow [x_1, x_2, x_3, x_4]$ $seed \leftarrow \{C_1, C_2, C_3, C_4\}$: UNSAT shrink $\rightarrow$ MUS : $\{C_1, C_2, C_3\}$ $Map \leftarrow Map \wedge (\neg x_1 \vee x_2 \vee x_3)$
Itération 2	$Map = (\neg x_1 \vee x_2 \vee x_3)$: SAT getModel $\rightarrow [\neg x_1, x_2, x_3, x_4]$ $seed \leftarrow \{C_2, C_3, C_4\}$: SAT grow $\rightarrow$ MSS : $\{C_2, C_3, C_4\}$, le MCS équivalent : $\{C_1\}$ $Map \leftarrow Map \wedge (x_1)$
Itération 3	$Map = (\neg x_1 \vee x_2 \vee x_3) \wedge (x_1)$: SAT getModel $\rightarrow [x_1, \neg x_2, x_3, x_4]$ $seed \leftarrow \{C_1, C_3, C_4\}$: UNSAT shrink $\rightarrow$ MUS : $\{C_1, C_4\}$ $Map \leftarrow Map \wedge (\neg x_1 \vee \neg x_4)$
Itération 4	$Map = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1) \wedge (\neg x_1 \vee \neg x_4)$: SAT getModel $\rightarrow [x_1, \neg x_2, x_3, \neg x_4]$ $seed \leftarrow \{C_1, C_3\}$: SAT grow $\rightarrow$ MSS : $\{C_1, C_3\}$, le MCS équivalent est $\{C_2, C_4\}$ $Map \leftarrow Map \wedge (x_2 \vee x_4)$
Itération 4	$Map = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_2 \vee x_4)$: SAT getModel $\rightarrow [x_1, x_2, \neg x_3, \neg x_4]$ $seed \leftarrow \{C_1, C_2\}$: SAT grow $\rightarrow$ MSS : $\{C_1, C_2\}$, le MCS équivalent est $\{C_3, C_4\}$ $Map \leftarrow Map \wedge (x_3 \vee x_4)$
Itération 4	$Map = (\neg x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1) \wedge (\neg x_1 \vee \neg x_4) \wedge (x_3 \vee x_4)$: SAT

TABLE 3.1 – Un exemple de déroulement de l'algorithme MARCO.

Exemple [Liffiton 2015] Soit l'ensemble de contraintes infaisable $C = \{C_1 = (a), C_2 = (\neg a \vee b), C_3 = (\neg b), C_4 = (\neg a)\}$ qui contient deux MUSs et trois MCSs : $\{MUSs = \{C_1, C_2\}, \{C_1, C_3, C_4\}\}$, $MCSs = \{\{C_1\}, \{C_2, C_4\}, \{C_2, C_3\}\}$. Dans la figure 3.7, nous représentons le diagramme de Hasse représentant l'ensemble des parties de l'ensemble C comme un treillis. Chaque niveau dans ce diagramme contient des sous-ensembles d'une certaine taille et les arcs relient les ensembles avec leurs surensembles immédiats et sous-ensembles.

La table 3.1 déroule l'algorithme MARCO sur l'ensemble de contraintes C pour trouver les MUSs et MCSs.

Chaque sous-ensemble dans $P(C)$ est soit satisfiable ou insatisfiable. Pour illustrer les sous-ensembles satisfiables ou insatisfiables sur le diagramme de Hasse de $P(C)$, nous allons colorer les noeuds satisfiables par le vert et les noeuds insatisfiables par le rouge, voir la figure 3.8. Dans la région verte, les noeuds des MSSs sont

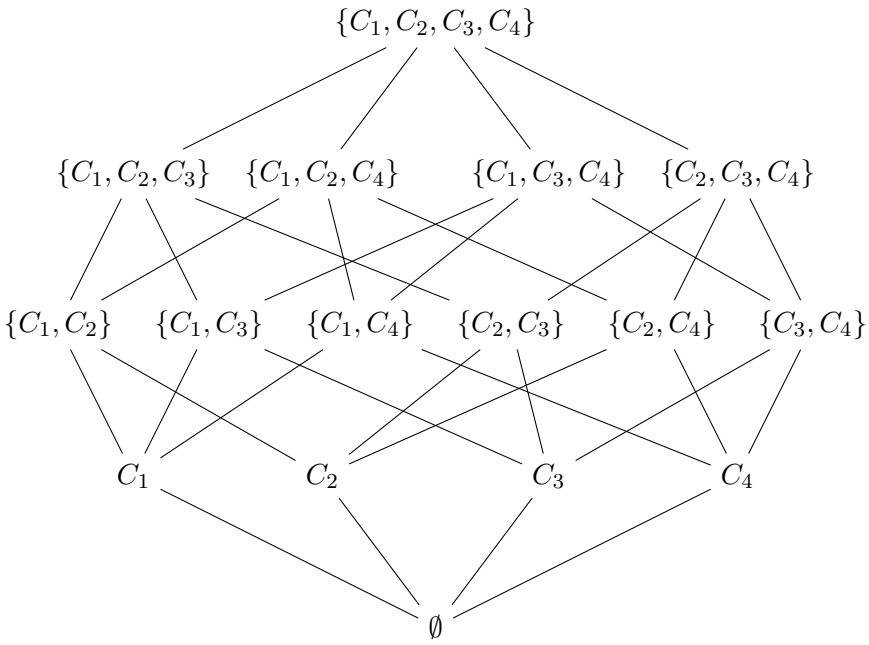


FIGURE 3.7 – Le diagramme de Hasse représentant l'ensemble des sous-ensembles de l'ensemble contraintes $C = \{C_1, C_2, C_3, C_4\}$.

en haut (par définition un MSS est maximal). Les noeuds des MUSs sont en bas de la région rouge (un MUS est par définition minimal).

3.6.4 Conclusion

DAA exploite aussi la relation entre MCSs et MUSs, comme l'approche CAMUS [Liffiton 2008]. La différence principale entre les deux est que CAMUS commence par calculer tous les MCSs, puis il calcule tous les MUSs(les couvertures minimales) ; alors que DAA calcule les deux tout au long de son exécution : DAA calcule la couverture minimale des MCSs obtenus après chaque nouveau MCS calculé. Selon [Bailey 2005], DAA dépasse les performances de l'approche basée sur l'énumération des sous-ensembles de Banda et al. dans [de la Banda 2003]. CAMUS souffre du fait que le nombre des MCSs peut être exponentiel. Cela a un impact négatif pour faire des progrès sur le calcul des MUSs, car le processus du calcul des MUSs est lancé après que les MCSs sont tous calculés. DAA est mieux, mais il ne pourrait pas trouver un MUS de taille k s'il n'a pas trouvé au moins k MCSs. Les expérimentations rapportées dans [Liffiton 2013] montrent que CAMUS peut énumérer l'ensemble de tous les MUSs plus rapidement que MARCO, tandis que MARCO est plus approprié pour le calcul de certains MUSs rapidement.

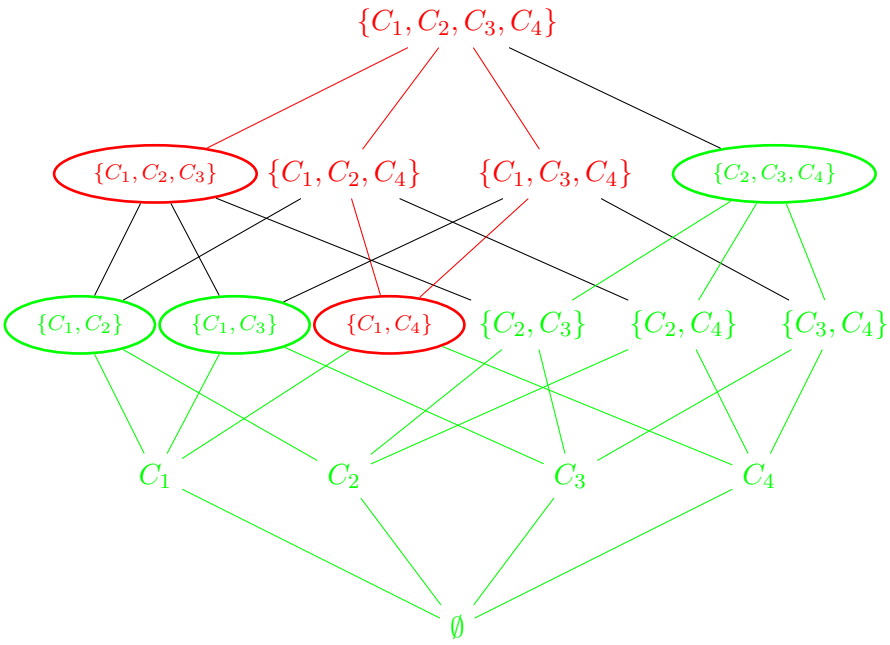


FIGURE 3.8 – Le diagramme de Hasse coloré indiquant les sous-ensembles SAT (colorés en vert) et UNSAT (colorés en rouge) dans $P(C)$, où $C = \{C_1, C_2, C_3, C_4\}$.

3.7 Conclusion

Un ensemble de contraintes peut être inconsistant (ou dit sur-contraint) si les contraintes ne peuvent pas être satisfaites simultanément [Meseguer 2003]. Les MCSs et MUSs sont des outils nécessaires pour comprendre pourquoi le système de contraintes est inconsistant. Le sujet de ma thèse consiste à chercher la localisation des erreurs dans un programme impératif à partir d'un contre-exemple. Ce problème est difficile, car la trace d'exécution du contre-exemple que l'utilisateur doit analyser contient une quantité très importante d'information. Notre intuition pour résoudre ce problème est de trouver les ensembles minimaux d'instructions suspectes dans cette trace qui explique pourquoi le programme a échoué. Pour cela, nous le traduisons en contraintes avec la postcondition et le contre-exemple pour obtenir un système qui est inconsistant. Il suffit ensuite d'isoler les MCSs et MUSs dans le système obtenu pour trouver les ensembles recherchés. En pratique, il est plus facile de trouver des sous-ensembles satisfiables de contraintes que des sous-ensembles insatisfiables. Donc, trouver les MCSs (équivalent à trouver leurs MSSs complémentaires) est plus facile que de trouver directement les MUSs [Liffiton 2008]. Dans notre approche de localisation d'erreurs, nous avons exporté la première phase de CAMUS calculant les MCSs, étudiée dans ce chapitre, pour chercher les MCSs de taille bornée dans le programme à partir du contre-exemple.

À travers ce chapitre, nous avons essayé de présenter les approches les plus connues pour diagnostiquer un système de contraintes inconsistant. Nous avons décrit

des algorithmes qui calculent tous les sous-ensembles de conflit minimes et des sous-ensembles de correction minimes.

Deuxième partie

Approche `LocFaults`

Cette partie est dédiée à présenter mes principales contributions à la résolution de la localisation d'erreurs dans un programme à partir d'un contre-exemple par l'utilisation des contraintes. Dans cette partie, nous détaillons les algorithmes développés et utilisés. Nous décrivons notre approche `LocFaults`*. Nous étudions également le traitement des boucles erronées, notamment le traitement des boucles avec le Bug* `Off-by-one`*.*

Chapitre 4

Une approche bornée à base de contraintes pour l'aide à la localisation d'erreurs

> *"Beware of bugs in the above code ; I have only proved it correct, not tried it."* – Donald Knuth

Sommaire

4.1 Introduction

L'aide à la localisation d'erreurs à partir de contre-exemples ou de traces d'exécution est une question cruciale lors de la mise au point de logiciels critiques. En effet, quand un programme P contient des erreurs, un Model Checker fournit un contre-exemple ou une trace d'exécution qui est souvent longue et difficile à comprendre, et de ce fait d'un intérêt très limité pour le programmeur qui doit débugger son programme. La localisation des portions de code qui contiennent des erreurs est donc souvent un processus difficile et coûteux, même pour des programmeurs expérimentés.

Dans ce chapitre, nous présentons notre nouvelle approche basée sur la programmation par contraintes (PPC, ou CP pour Constraint Programming en anglais) pour l'aide à la localisation des erreurs dans un programme pour lequel un contre-exemple

a été trouvé ; c'est à dire pour lequel on dispose d'une instanciation des variables d'entrée qui viole la postcondition. Pour aider à localiser les erreurs, nous générons un système de contraintes pour les chemins du CFG (Graphe de Flot de Contrôle) où au plus k instructions conditionnelles sont susceptibles de contenir des erreurs. Puis, nous calculons pour chacun de ces chemins des ensembles minima de correction (ou MCSs - Minimal Correction Sets) de taille bornée. Le retrait d'un de ces ensembles des contraintes initiales produit un MSS (Maximal Satisfiable Subset) qui qui rend le système considéré satisfiable. Nous adaptons pour cela un algorithme proposé par Liffiton et Sakallah (voir alg. 9) afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques.

Ce chapitre est structuré de la façon suivante. La section 4.2 illustre comment `LocFaults` travaille sur un programme simple. Nous introduirons la notation $\leq k$-DCM dans la section 4.3. La section 4.4 présente notre approche en détaillant les différents algorithmes utilisés. Nous expliquons notre contribution pour le traitement des boucles erronées, notamment le bug *Off-by-one*, dans la section 4.5. La section 4.6 parle de la conclusion du chapitre.

4.2 Exemple explicatif

Considérons le programme `AbsMinus` (voir fig. 4.1). Les entrées sont des entiers $\{i,\ j\}$ et la sortie attendue est la valeur absolue de $i-j$. Une erreur a été introduite sur la ligne 10, ainsi pour les données d'entrée $\{i=0, j=1\}$, `AbsMinus` retourne -1. La post-condition (notée $POST$) est $result = |i-j|$.

```
1  class AbsMinus {
2  /*Il renvoie |i-j|, la valeur absolue de i moins j*/
3  /*@ ensures
4   @ ((i < j) ==> (\result == j-i)) &&
5   @ ((i >= j) ==> (\result == i-j)); */
6   int AbsMinus (int i, int j) {
7     int result;
8     int k = 0;
9     if (i <= j) {
10     k = k+2; }  // erreur: k = k + 2 au lieu de k = k + 1
11    if (k == 1 && i != j) {
12    result = j-i; }
13    else {
14    result = i-j; }
15    return result;  } }
```

FIGURE 4.1 – Le programme AbsMinus

Le graphe de flot de contrôle (CFG) du programme `AbsMinus` et un chemin erroné sont représentés dans la figure 4.2. Ce chemin erroné correspond aux données d'entrée : $\{i=0, j=1\}$. Tout d'abord, `LocFaults` collecte sur le chemin 4.2.(b) l'ensemble de contraintes $C_1 = \{i_0 = 0, j_0 = 1, k_0 = 0, k_1 = k_0 + 2, r_1 = i_0 - j_0\}$ [1]. Puis, `LocFaults` calcule les MCSs de $C_1 \cup POST$. Seulement un MCS peut être trouvé : $\{r_1 = i_0 - j_0\}$. En d'autres termes, si nous supposons que les instructions

1. Nous utilisons la transformation en forme DSA [Barnett 2005] qui assure que chaque variable est affectée une seule fois sur chaque chemin du CFG.

conditionnelles sont correctes, la seule instruction suspecte sur ce chemin erroné est l'instruction 14.

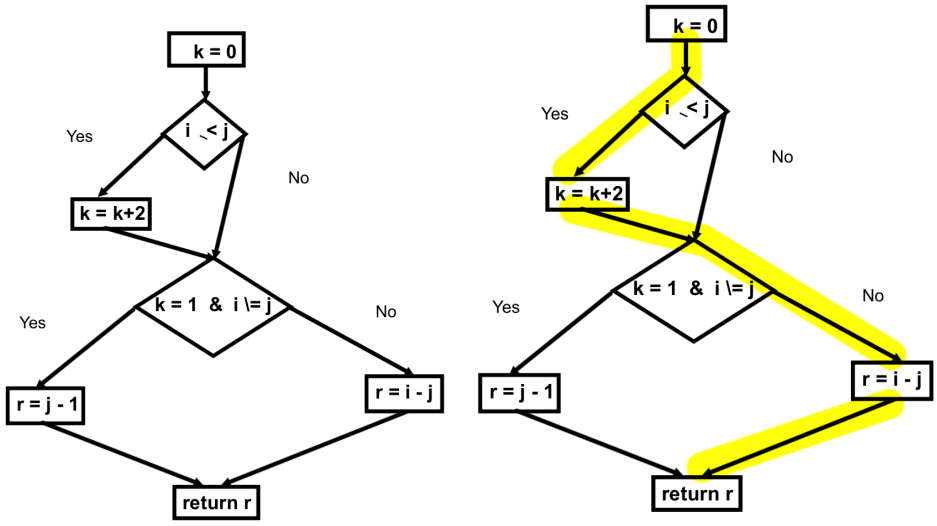


FIGURE 4.2 – Le CFG et chemin erroné – Le programme `AbsMinus`

Ensuite, `LocFaults` commence le processus de déviation. La première déviation (voir la figure 4.3.(a), chemin vert) produit encore un chemin qui viole la post-condition, et donc, nous l'ignorons. La seconde déviation (voir la figure 4.3.(b), chemin bleu) produit un chemin qui satisfait la postcondition. Donc, `LocFaults` collecte les contraintes sur la partie du chemin 4.3.(b) qui précède la condition déviée, en d'autres mots $C_2 = \{i = 0, j = 1, k_0 = 0, k_1 = k_0 + 2\}$. Puis `LocFaults` recherche les MCS de $C_2 \cup \neg(k = 1 \wedge i \neq j)$; c'est-à-dire nous essayons d'identifier les instructions qui doivent être modifiées afin que le programme prenne un chemin satisfaisant la post-condition pour les données d'entrée. Ainsi, pour cette deuxième déviation deux instructions suspectes sont identifiées :

- L'instruction conditionnelle sur la ligne 11 ;
- L'affectation sur la ligne 10 car la contrainte correspondante est le seul MCS dans $C_2 \cup \neg(k = 1 \wedge i \neq j)$.

Puis, `LocFaults` tente de dévier une seconde condition. Le seul chemin possible est celui où les deux conditions du programme `AbsMinus` sont déviées. Cependant, comme il a le même préfixe que le premier chemin dévié, nous le rejetons.

4.3 Notation $\leq k$-DCM

Soit un programme erroné modélisé en un CFG $G = (C, A, E)$: C est l'ensemble des nœuds conditionnels, A est l'ensemble des blocs d'affectation, E est l'ensemble des arcs, et un contre-exemple. Une DC (*Déviation de Correction*) est un ensemble

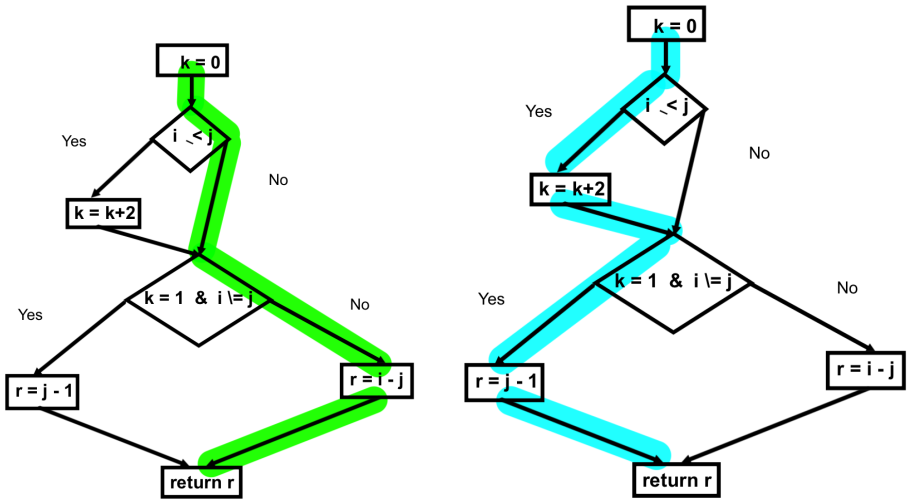


FIGURE 4.3 – Les chemins avec une déviation – Le programme `AbsMinus`

$D \subseteq C$ telle que la propagation du contre-exemple sur l'ensemble des instructions de G à partir de la racine, tout en ayant nié chaque condition [2] dans D, permet en sortie de satisfaire la postcondition. Une DC est dite minimale (DCM pour *Déviation de Correction* Minimale), ou irréductible, dans le sens où aucun élément ne peut être retiré de D sans que celle-ci ne perde cette propriété. En d'autres termes, D est une correction minimale du programme dans l'ensemble des conditions. La taille d'une déviation minimale est son cardinal. Le problème $\leq k$-*DCM* consiste à trouver toutes les DCMs de taille inférieure ou égale à k.

Exemple, le CFG du programme AbsMinus (voir fig. 4.2) possède une déviation minimale de taille 1 pour le contre-exemple $\{i = 0, j = 1\}$. Certes, la déviation $\{i \leq j, k = 1 \wedge i \neq j\}$ permet de corriger le programme, mais elle n'est pas minimale ; la seule déviation minimale pour ce programme est $\{k = 1 \wedge i \neq j\}$.

4.4 Notre approche

Dans cette section nous allons d'abord présenter le cadre général de notre approche, à savoir celui du "Bounded Model Checking" (BMC) basé sur la programmation par contraintes et comment nous calculons les MCSs d'une taille limitée sur les chemins du programme. Puis, nous allons décrire la méthode proposée et les algorithmes utilisés : nous détaillons comment nous calculons les MCSs d'une taille bornée sur les chemins du programme, en utilisant une recherche en profondeur (ou DFS, pour Depth First Search) sur le CFG du programme. Enfin, nous illustrons comment `LocFaults` se déroule en traçant son exécution sur un exemple simple.

2. On nie la condition afin de prendre la branche opposée à celle où on devait aller.

4.4.1 BMC à base de contraintes

Notre approche se place dans le cadre du "Bounded model Checking" (BMC) par programmation par contraintes [Collavizza 2010, Collavizza 2014]. En BMC, les programmes sont dépliés en utilisant une borne b, c'est à dire que les boucles sont remplacées par des imbrications de conditionnelles de profondeur au plus b. Il s'agit ensuite de détecter des non-conformités par rapport à une spécification. Étant donné un triplet de Hoare $\{PRE,PROG_b,POST\}$, où PRE est la pré-condition, $PROG_b$ est le programme déplié b fois et $POST$ est la post-condition, le programme est *non conforme* si la formule $\Phi = PRE \wedge PROG_b \wedge \neg POST$ est satisfiable. Dans ce cas, une instanciation des variables de Φ est un *contre-exemple*, et un cas de non conformité, puisqu'il satisfait à la fois la pré-condition et le programme, mais ne satisfait pas la post-condition.

`CPBPV` [Collavizza 2010] est un outil de BMC basé sur la programmation par contraintes.`CPBPV` transforme PRE et $POST$ en contraintes, et transforme $PROG_b$ en un CFG dans lequel les conditions et les affectations sont traduites en contraintes[3]. `CPBPV` construit le `CSP` de la formule Φ *à la volée*, par un parcours en profondeur du graphe. À l'état initial, le `CSP` contient les contraintes de PRE et $\neg POST$, puis les contraintes d'un chemin sont ajoutées au fur et à mesure de l'exploration du graphe. Quand le dernier noeud d'un chemin est atteint, la faisabilité du `CSP` est testée. S'il est consistant, alors on a trouvé un contre-exemple, sinon, un retour arrière est effectué pour explorer une autre branche du CFG. Si tous les chemins ont été explorés sans trouver de contre-exemple, alors le programme est conforme à sa spécification (sous réserve de l'hypothèse de dépliage des boucles).

4.4.2 MCSs pour localiser les erreurs

Les travaux présentés cherchent à *localiser* les erreurs détectées par la phase de BMC. Plus précisément, soit CE une instanciation des variables qui satisfait le `CSP` contenant les contraintes de PRE et $\neg POST$, et les contraintes d'un chemin incorrect de $PROG_b$ noté $PATH$. Alors le `CSP` $C = CE \cup PRE \cup PATH \cup POST$ est *inconsistant*, puisque CE est un contre-exemple et ne satisfait donc pas la post-condition. Un *ensemble minima de correction* (ou MCS - Minimal Correction Set) de C est un ensemble de contraintes qu'il faut nécessairement enlever pour que C devienne consistant (voir la section 3.2). Un tel ensemble fournit donc une *localisation de l'erreur* sur le chemin du contre-exemple. Comme l'erreur peut se trouver dans une affectation sur le chemin du contre-exemple, mais peut aussi provenir d'un mauvais branchement, notre approche (nommée `LocFaults`) s'intéresse également aux MCSs des systèmes de contraintes obtenus en déviant des branchements par rapport au comportement induit par le contre-exemple. Plus précisément, l'algorithme `LocFaults` effectue un parcours en profondeur d'abord du CFG de $PROG_b$,

3. Pour éviter les problèmes de re-définitions multiples des variables, la forme DSA (Dynamic Single Assignment [Barnett 2005]) est utilisée

en propageant le contre-exemple et en déviant au plus b_{cond} conditions. Trois cas peuvent être distingués :

- Aucune condition n'a été déviée : `LocFaults` a parcouru le chemin du contre-exemple en collectant les contraintes de ce chemin et il va calculer les MCSs sur cet ensemble de contraintes ;
- b_{cond} conditions ont été déviées sans qu'on arrive à trouver un chemin qui satisfasse la post-condition : on abandonne l'exploration de ce chemin ;
- d conditions on déja été déviées et on peut encore dévier au moins une condition, c'est à dire $b_{cond} > 1$. Alors la condition courante c est déviée. Si le chemin résultant ne viole plus la post-condition, l'erreur sur le chemin initial peut avoir deux causes différentes :
  (i) les conditions déviées elles-mêmes sont cause de l'erreur,
  (ii) une erreur dans une affectation a provoqué une mauvaise évaluation de c, faisant prendre le mauvais branchement.
  Dans le cas (ii), le `CSP` $CE \cup PRE \cup PATH_c \cup \{c\}$, où $PATH_c$ est l'ensemble des contraintes du chemin courant, c'est à dire le chemin du contre-exemple dans lequel d déviations ont déjà été prises, est satisfiable. Par conséquent, le `CSP` $CE \cup PRE \cup PATH_c \cup \{\neg c\}$ est *insatisfiable*. `LocFaults` calcule donc également les MCSs de ce `CSP`, afin de détecter les instructions suspectées d'avoir induit le mauvais branchement pour c.
  Bien entendu, nous ne calculons pas les MCSs des chemins ayant le même préfixe : si la déviation d'une condition permet de satisfaire la postcondition, il est inutile de chercher à modifier des conditions supplémentaires dans la suite du chemin

4.4.3 Description de l'algorithme

L'algorithme `LocFaults` (cf. Algorithm 15) prend en entrée un programme déplié non conforme vis-à-vis de sa spécification, un contre-exemple, b_{cond} : une borne sur le nombre de conditions qui sont déviées, et b_{mcs} : une borne sur le nombre de MCSs (Minimal Correction Subsets) générés. Il dévie au plus b_{cond} conditions par rapport au contre-exemple fourni, et renvoie une liste de corrections possibles. Pour cela, il procède de façon incrémentale. Il dévie successivement de 0 à b_{cond} conditions et il recherche les MCSs pour les chemins correspondants. Toutefois, si à l'étape k `LocFaults` a dévié une condition c_i et que cela a corrigé le programme, il n'explorera pas à l'étape k' avec $k' > k$ les chemins qui impliquent une déviation de la condition c_i. La cardinalité de la déviation minimale trouvée (k) est ajoutée comme information sur le nœud de c_i.

4.4.3.1 Exploration du CFG

`LocFaults` commence par construire le CFG du programme puis appelle la fonction `DFS` sur le chemin du contre-exemple (i.e. en déviant 0 condition) puis en accep-

tant au plus b_{cond} déviations. La fonction `DFS` gère trois ensembles de contraintes :

- CSP_d : l'ensemble des contraintes des conditions qui ont été déviées à partir du chemin du contre-exemple,
- CSP_a : l'ensemble des contraintes d'affectations du chemin,
- P : l'ensemble des contraintes dites de propagation, c'est à dire les contraintes de la forme $variable = constante$ qui sont obtenues en propageant le contre exemple sur les affectations du chemin.

L'ensemble P est utilisé pour propager les informations et vérifier si une condition est satisfaite, les ensembles CSP_d et CSP_a sont utilisés pour calculer les MCSs. Ces trois ensembles sont collectés à la volée lors du parcours en profondeur. Les paramètres de la fonction `DFS` sont les ensembles CSP_d, CSP_a et P décrits ci-dessus, n le noeud courant du CFG, MCS la liste des corrections en cours de construction, k_c le nombre de conditions déjà déviées sur le chemin courant, k le nombre de déviations autorisées, b_{mcs} la borne de la taille des MCS et $POST$ la postcondition utilisée. Nous notons $n.left$ (resp. $n.right$) la branche *if* (resp. *else*) d'un noeud conditionnel, et $n.next$ le noeud qui suit un bloc d'affectation ; *cstr* est la fonction qui traduit une condition ou affectation en contraintes.

Le parcours commence avec CSP_d et CSP_a vides et P contenant les contraintes du contre-exemple. Il part de la racine de l'arbre ($CFG.root$) qui contient la pré-condition, et se déroule comme suit :

* Il propage premièrement CE sur le CFG jusqu'à la fin du chemin initial erroné. Puis, il calcule au plus b_{mcs} MCSs sur le CSP courant. Ce qui représente une première localisation sur le chemin du contre-exemple.
* `LocFaults` essaye ensuite de dévier une condition. Lorsque le premier nœud conditionnel (noté *cond*) est atteint, `LocFaults` prend la décision opposée à celle induite par CE, et continue à propager CE jusqu'au dernier nœud dans le CFG. On utilise P pour savoir si la condition est satisfaite. On vérifie si cette déviation corrige le programme en appelant la fonction *collect*. Cette fonction propage tout simplement le contre-exemple sur le graphe à partir du noeud courant et renvoie les contraintes de propagation du chemin induit par P jusqu'à la postcondition. Si l'ensemble de contraintes construit à partir du chemin dévié satisfait la postcondition, il y a deux types d'ensemble d'instructions suspectes :
  - le premier est la condition *cond* elle-même. En effet, changer la décision pour *cond* permet à CE de satisfaire la postcondition ;
  - une autre cause possible de l'erreur est une ou plusieurs mauvaises affectations avant *cond* qui ont produit une décision erronée. Puis, `LocFaults` calcule aussi au plus b_{mcs} MCSs sur le CSP qui contient les contraintes collectées sur le chemin qui arrivent à *cond*.

  Ce processus est répété sur chaque nœud conditionnel du chemin du contre-exemple.
* Un processus similaire est ensuite appliqué pour dévier pour tout $k \leq b_{cond}$ conditions. Pour améliorer l'efficacité, les nœuds conditionnels qui corrigent le

programme sont marqués avec le nombre de déviations qui ont été faites avant d'avoir été atteints. Pour une étape donnée k, si le changement de la décision d'un nœud conditionnel *cond* marqué avec la valeur k' avec $k' \leq k$ corrige le programme, cette condition est ignorée. En d'autres termes, nous considérons seulement la première fois où un nœud conditionnel corrige le programme.

Algorithm 15: LocFaults

```
1  Fonction LocFaults(PROG,CE,POST,b_cond,b_mcs)
   Entrées:
   – PROG : un programme non conforme vis-à-vis de sa spécification ne contenant pas de boucles (i.e.
     programme sans boucle ou après dépliage des boucles),
   – POST : postcondition du programme PROG,
   – CE : un contre-exemple de PROG,
   – b_cond : le nombre maximum de conditions à dévier,
   – b_mcs : la cardinalé maximale des MCS
   Sorties: une liste de MCS de cardinalité ≤ b_mcs pour chacun des chemins explorés
2  début
3      CFG ← CFG_build(PROG) % construction du CFG
4      MCS = ∅
5      % calcul des MCS sur le chemin du contre-exemple
6      CSP_a ← ∅ % CSP_a : contraintes des affectations collectées par DFS_devie
7      DFS_devie(CFG.root,CE,POST,CSP_a,∅,0, 0,MCS,b_mcs)
8      MCS.add(MCS(CSP_a ∪ cstr(Post), MSC_b))
9      % calcul des MCS sur les chemins ayant de 1 à b_cond déviations
10     pour k = 1 ... b_cond faire
11         DFS_devie(CFG.root,CE,POST,∅,∅,0, k,MCS,b_mcs)
12     fin
13     retourner MCS
14 fin
```

4.4.3.2 Calcul des MCSs

Pour calculer les MCSs, `LocFaults` appelle l'algorithme `MCS` (cf. Algorithm 18) qui est une transcription directe de l'algorithme proposé par Liffiton et Sakallah [Liffiton 2008]. Cet algorithme prend en entrée l'ensemble de contraintes infaisable qui correspond au chemin identifié (C), et b_{mcs} : la borne sur la taille des MCSs calculés. Chaque contrainte c_i dans le système construit C est augmentée par un indicateur y_i pour donner $y_i \rightarrow c_i$ dans le nouveau système de contraintes C'. Affecter à y_i la valeur $Vrai$ implique la contrainte c_i ; en revanche, affecter à y_i la valeur $Faux$ implique la suppression de la contrainte c_i. Un MCS est obtenu en cherchant une affectation qui satisfait le système de contraintes avec un ensemble minimal d'indicateurs de contraintes affectés à $Faux$. Pour limiter le nombre de variables indicateurs de contraintes qui peuvent être assignées à Faux, on utilise la contrainte $AtMost(\neg y_1, \neg y_2, ..., \neg y_n, k)$ (voir la ligne 5), le système créé est noté dans l'algorithme C'_k (ligne 5). Chaque itération de la boucle WHILE (lignes $6-19$) permet de trouver tous les MCSs de taille k, k est incrémenté de 1 après chaque itération. Après chaque MCS trouvé (lignes $8-13$), une contrainte de blocage est ajoutée à C'_k et C' pour empêcher de trouver ce nouveau MCS dans les prochaines itérations (lignes $15-16$). La procédure `BlockingClause(newMCS)` appelée à la ligne 15 (resp. ligne 16) permet d'exclure les sur-ensembles de taille k (resp. de taille supérieure à k). La première boucle (lignes $4-19$) s'itère jusqu'à ce que tous les MCSs de C

Algorithm 16: DFS_{devie}

```
1  Procedure DFS_devie(n,P,POST,CSP_d,CSP_a,k_c,k,MCS,b_mcs)
   Entrées:
   – n : noeud du CFG,
   – n.sd : k si la déviation de n est la kème déviation sur un chemin qui satisfait la post-condition ; sinon -1
   – P : contraintes de propagation (issues du contre-exemple et du chemin),
   – POST : la postcondition utilisée,
   – CSP_d : contraintes des conditions déviées,
   – CSP_a : contraintes des affectations,
   – k_c : nombre de conditions déjà déviées sur le chemin courant,
   – k : nombre de conditions à dévier,
   – MCS : ensemble des MCS calculés,
   – b_mcs : la borne du cardinal des MCS
2  début
3      si n est un noeud conditionnel alors
4          si P ∪ {cstr(n.cond)} est faisable alors
5              % next est le noeud où l'on doit aller, devie est la branche opposée
6              next = n.gauche
7              devie = n.droite
8          fin
9          sinon
10             next = n.droite
11             devie = n.gauche
12         fin
13         si kc = k − 1 and (n.sd = −1 or kc < n.sd) alors
14             % Il ne reste plus qu'une condition à dévier sur le chemin courant et
15             % - soit on n'a jamais dévié avec succès la condition,
16             % - soit cette condition a déjà été déviée avec succès sur un chemin impliquant plus
                 de kc déviations,
17             % on essaie donc de dévier la condition du noeud n
18             si collect(devie,P) ∪ {cstr(Post)} est faisable alors
19                 % le chemin satisfait la postcondition : on met à jour sd et les MCS
20                 n.sd = k_c + 1
21                 CSP_d ← CSP_d ∪ {cstr(n.cond)}
22                 MCS.addAll(CSP_d) % ajout des conditions déviées
23                 % calcul des MCS sur le chemin qui mène à la dernière condition déviée
24                 pour chaque c dans CSP_d faire
25                     CSP_a ← CSP_a ∪ {¬c}
26                 fin
27                 MCS.add(MCS(CSP_a, b_mcs))
28             fin
29         fin
30         sinon si kc < k − 1 and (n.sd = −1 or kc < n.sd) alors
31             % on essaie de dévier la condition courante et des conditions en dessous
32             DFS_devie(devie,P,POST,CSP_d ∪ {cstr(n.cond)},CSP_a,k_c + 1,k,MCS,b_mcs)
33         fin
34         % dans tous les cas, on essaie de suivre le chemin courant et dévier des conditions en
             dessous du noeud n
35         DFS_devie(next,P,POST,CSP_d,CSP_a,k_c,k,MCS,b_mcs)
36     fin
37     sinon si (n est un bloc d'affectations) alors
38         pour chaque affectation ass ∈ n.assigns faire
39             P.add(propagate(ass,P))
40             CSP_a ← CSP_a ∪ {cstr(ass)}
41         fin
42         % On continue l'exploration sur le noeud suivant
43         DFS_devie(n.next,P,POST,CSP_d,CSP_a,k_c,k,MCS,b_mcs)
44     fin
45     % Si n est le noeud de la postcondition, on a terminé l'exploration du chemin courant
46 fin
```

Algorithm 17: collect

```
1  Fonction collect(n,P)
   Entrées:
   – n : noeud du CFG,
   – P : contraintes de propagation du chemin jusqu'à n,
   Sorties:
   – P : contraintes de propagation du chemin induit par P en entrée jusqu'à la post-condition,
2  début
3      si n est la postcondition alors
4          retourner P
5      fin
6      sinon si n est un noeud conditionnel alors
7          si P ∪ {cstr(n.cond)} est faisable alors
8              % exploration de la branche If
9              retourner collect(n.left,P)
10         fin
11         sinon
12             retourner collect(n.right,P)
13         fin
14     fin
15     sinon si (n est un bloc d'affectations) alors
16         % on propage les affectations
17         pour chaque affectation ass ∈ n.assigns faire
18             P.add(propagate(ass,P))
19         fin
20         % On continue l'exploration sur le noeud suivant
21         retourner collect(n.next,P)
22     fin
23 fin
```

soient générés (C' devient infaisable) ; elle peut également s'arrêter si les MCSs de taille inférieure ou égale b_{mcs} sont obtenus ($k > b_{mcs}$).

4.4.4 Exemple

Nous allons sur un exemple illustrer le déroulement du marquage de notre approche, voir le graphe sur la figure 4.4. Chaque cercle dans le graphe représente un nœud conditionnel visité par l'algorithme. L'exemple ne montre pas les blocs d'affectations, car nous voulons illustrer uniquement comment nous trouverons les déviations de correction minimales d'une taille bornée de la manière citée ci-dessus. Un arc reliant une condition c_1 à une autre c_2 illustre que c_2 est atteinte par l'algorithme. Il y a deux façons, par rapport au comportement du contre-exemple, par lesquelles `LocFaults` arrive à la condition c_2 :

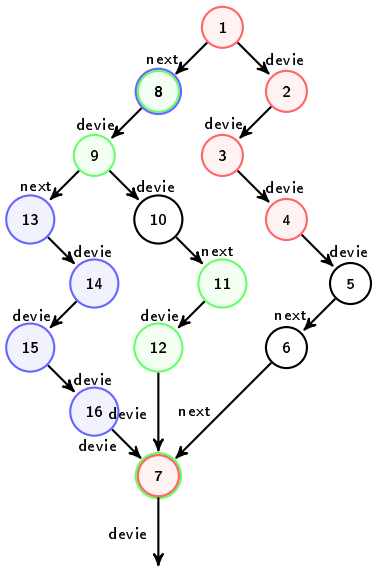
1. en suivant la branche normale induite par la condition c_1 ;

2. en suivant la branche opposée.

La valeur de l'étiquette des arcs pour le cas (1) (resp. (2)) est "*next*" (resp. "*devie*").

Algorithm 18: MCS

```
1  Fonction MCS(C,b_mcs)
   Entrées: C : Ensemble de contraintes infaisable, b_mcs : Entier
   Sorties: MCS : Liste de MCSs de C de cardinalité inférieure à b_mcs
2  début
3      C' ← AddYVars(C) ; MCS ← ∅ ; k ← 1 ;
4      tant que SAT(C') ∧ k ≤ b_mcs faire
5          C'_k ← C' ∧ AtMost({¬y_1, ¬y_2, ..., ¬y_n},k)
6          tant que SAT(C'_k) faire
7              newMCS ← ∅
8              pour chaque indicateur y_i faire
9                  % y_i est l'indicateur de la contrainte c_i ∈ C, et val(y_i) la
                   valeur de y_i dans la solution calculée de C'_k.
10                 si val(y_i) = 0 alors
11                     newMCS ← newMCS ∪ {c_i}.
12                 fin
13             fin
14             MCS.add(newMCS).
15             C'_k ← C'_k ∧ BlockingClause(newMCS)
16             C' ← C' ∧ BlockingClause(newMCS)
17         fin
18         k ← k + 1
19     fin
20     retourner MCS
21 fin
```



le chemin $< 1, 2, 3, 4, 5, 6, 7, ..., POST >$ est correct
le chemin $< 1, 8, 9, 10, 11, 12, 7, ..., POST >$ est correct

FIGURE 4.4 – Figure illustrant l'exécution de notre algorithme sur un exemple pour lequel deux déviations minimales sont détectées : $\{1, 2, 3, 4, 7\}$ et $\{8, 9, 11, 12, 7\}$, et une abandonnée : $\{8, 13, 14, 15, 16, 7\}$. Sachant que la déviation de la condition "7" a permis de corriger le programme pour le chemin $< 1, 2, 3, 4, 5, 6 >$, ainsi que pour le chemin $< 1, 8, 9, 10, 11, 12, 7 >$.

- À l'étape $k = 5$, notre algorithme a identifié deux déviations minimales de taille égale à 5 :
  1. $D_1 = \{1, 2, 3, 4, 7\}$, le nœud "7" est marqué par la valeur 5 ;
  2. $D_2 = \{8, 9, 11, 12, 7\}$, elle a été autorisée, car la valeur de la marque du nœud "7" est égale à la cardinalité de D_2.
- À l'étape $k = 6$, l'algorithme a suspendu la déviation suivante $D_3 = \{8, 13, 14, 15, 16, 7\}$, car la cardinalité de D_3 est supérieure strictement à la valeur de l'étiquette du nœud "7".

4.5 Traitement des boucles

Dans le cadre du Bounded Model Checking (BMC) pour les programmes, le dépliage peut être appliqué au programme en entier comme il peut être appliqué aux boucles séparément [D'silva 2008]. Notre approche de localisation d'erreurs, `LocFaults` [Bekkouche 2014, Bekkouche 2015b], se place dans la deuxième démarche ; c'est-à-dire, nous utilisons une borne b pour déplier les boucles en les remplaçant par des imbrications de conditionnelles de profondeur b. Considérons le programme `Minimum` (voir fig. 4.5) contenant une seule boucle, qui calcule le minimum dans un tableau d'entiers. L'effet sur le graphe de flot de contrôle du

programme `Minimum` avant et après le dépliage est illustré sur les figures respectivement 4.5 et 4.6 : la boucle *While* est dépliée 3 fois, tel que 3 est le nombre d'itérations nécessaires à la boucle pour calculer la valeur minimum dans un tableau de taille 4 dans le pire des cas.

Parmi les erreurs les plus courantes associées aux boucles, nous nous intéressons au bug *Off-by-one* lorsque des boucles s'itèrent une fois de trop ou de moins. Cela peut être dû à une mauvaise initialisation des variables de contrôle de la boucle, ou à une condition incorrecte de la boucle. Le programme `Minimum` présente un cas de ce type d'erreur. Il est erroné à cause de sa boucle *While*, l'instruction falsifiée se situe sur la condition de la boucle (ligne 9) : la condition correcte doit être $(i < tab.length)$ ($tab.length$ est le nombre d'éléments du tableau tab). À partir du contre-exemple suivant : $\{tab[0] = 3, tab[1] = 2, tab[2] = 1, tab[3] = 0\}$, nous avons illustré sur la figure 4.6 le chemin fautif initial (voir le chemin coloré en rouge), ainsi que la déviation pour laquelle la postcondition est satisfaisable (la déviation ainsi que le chemin au-dessus de la condition déviée sont illustrés en vert).

```
1  class Minimum {
2  /* The minimum in an array of n integers
       */
3  /*@ ensures
4    @ (\forall int k;(k >= 0 && k < tab.
         length);tab[k] >= min);
5    @*/
6    int Minimum (int[] tab) {
7      int min=tab[0];
8      int i = 1;
9      while (i<tab.length-1) { /*error,
           the condition should be (i<tab.
           length)*/
10       if (tab[i]<=min){
11         min=tab[i];
12       }
13       i = i+1;
14     }
15     return min;
16   }
17 }
```

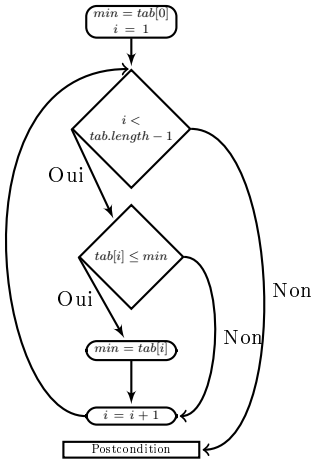


FIGURE 4.5 – Le programme `Minimum` et son CFG normal (non déplié). La postcondition est $\{\forall$ int k; $(k \geq 0 \wedge k < tab.length); tab[k] \geq min\}$

Nous affichons dans la table 4.1 les chemins erronés générés (la colonne $PATH$) ainsi que les MCSs calculés (la colonne $MCSs$) pour au plus 1 condition déviée par rapport au comportement du contre-exemple. La première ligne correspond au chemin du contre-exemple ; la deuxième correspond au chemin obtenu en déviant la condition $\{i_2 \leq tab_0.length - 1\}$.

`LocFaults` a permis d'identifier un seul MCS sur le chemin du contre-exemple qui contient la contrainte $min_2 = tab_0[i_1]$, l'instruction de la ligne 11 dans la deuxième itération de la boucle dépliée. Avec une condition déviée, l'algorithme suspecte la troisième condition de la boucle dépliée, $i_2 < tab_0.length - 1$; en d'autres termes, il faut une nouvelle itération pour satisfaire la postcondition.

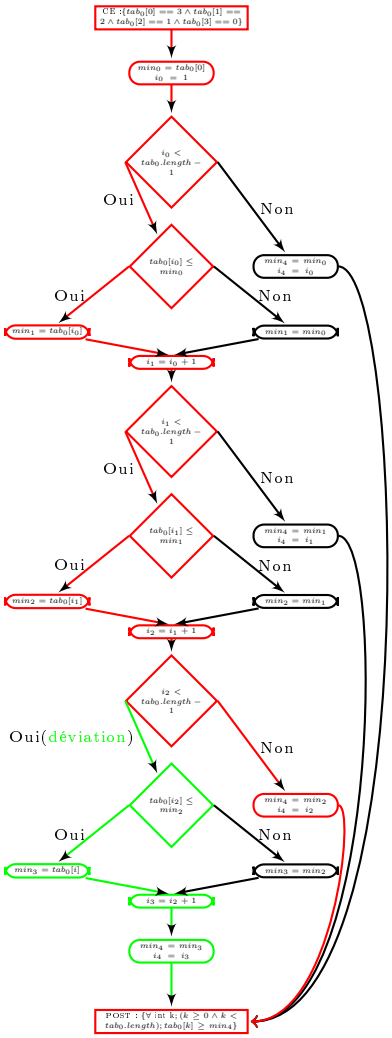


FIGURE 4.6 – Figure montrant le CFG en forme DSA du programme `Minimum` en dépliant sa boucle 3 fois, avec le chemin d'un contre-exemple (illustré en rouge) et une déviation satisfaisant sa postcondition (illustrée en vert).

Cet exemple illustre un cas d'un programme avec une boucle erronée : l'erreur est sur le critère d'arrêt, elle ne permet pas en effet au programme d'itérer jusqu'au dernier élément du tableau en entrée. `LocFaults` avec son mécanisme de déviation arrive à supporter ce type d'erreur avec précision. Il fournit à l'utilisateur non seulement les instructions suspectes dans la boucle non dépliée du programme original, mais aussi des informations sur les itérations où elles se situent concrètement en dépliant la boucle. Ces informations pourraient être très utiles pour le programmeur pour mieux comprendre les erreurs dans la boucle.

$PATH$	MCSs
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0],\ min_0 = tab_0[0],\ i_0 = 1,\ min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1],\ i_2 = i_1 + 1, min_3 = min_2,\ i_3 = i_2,\ POST : [(tab[0] \geq min_3) \wedge (tab[1] \geq min_3) \wedge (tab[2] \geq min_3) \wedge (tab[3] \geq min_3)]\}$	$\{min_2 = tab_0[i_1]\}$
$\{CE : [tab_0[0] = 3 \wedge tab_0[1] = 2 \wedge tab_0[2] = 1 \wedge tab_0[3] == 0],\ min_0 = tab_0[0],\ i_0 = 1,\ min_1 = tab_0[i_0], i_1 = i_0 + 1, min_2 = tab_0[i_1],\ i_2 = i_1 + 1, [\neg(i_2 \leq tab_0.length - 1)]$	$\{i_0 = 1\}$, $\{i_1 = i_0 + 1\}$, $\{i_2 = i_1 + 1\}$

TABLE 4.1 – Chemins et MCSs générés par LocFaults pour le programme `Minimum`.

4.6 Conclusion

Nous avons présenté dans ce chapitre notre nouvelle approche pour l'aide à la localisation d'erreurs dans un programme pour lequel un contre-exemple est disponible. Cette approche est basée sur la programmation par contraintes et dirigée par les flots. Elle permet de localiser les erreurs sur et autour du chemin du contre-exemple. Notre approche fournit à l'utilisateur des sous-exemples d'instructions dont le retrait de chacun corrige l'échec dans le programme.

Nous avons illustré sur un exemple simple comment `LocFaults` produit les instructions suspectes sur chaque chemin erroné en explorant le CFG du programme à partir du contre-exemple. Notre approche affiche les sous-ensembles minimaux corrigeant le programme séparément. Cela peut faciliter la tache de la compréhension et la correction des erreurs à l'utilisateur.

Nous avons présenté notre algorithme, `LocFaults`, qui explore en profondeur le CFG du programme déplié, en propageant le contre-exemple et en déviant au plus b_{cond} branchements conditionnels. Cette algorithme est incrémental. Il travaille par étapes successives : a une étape donnée de l'algorithme, si un noeud dans le CFG du programme a permis de détecter une DCM, il sera marqué par le cardinal de cette derniére, pour qu'aux prochaines étapes, l'algorithme abandonne toute déviation à partir de ce noeud. Après chaque DCM trouvée, `LocFaults` calcule les MCSs sur la partie du chemin qui précède la dernière condition déviée, le retrait d'au moins un de ces ensembles permet de suivre le chemin de la déviation (corriger le programme). Cet algorithme calcule les MCSs à l'aide de l'algorithme générique de Liffiton et Sakallah [Liffiton 2008] pour traiter d'une manière efficace les programmes avec calculs numériques.

Nous avons expliqué comment `LocFaults` trouve les erreurs dans les boucles. Les informations sur les itérations erronées fournies par `LocFaults` en plus des instructions suspectes (MCSs et DCMs calculés) pourraient aider l'utilisateur pour corriger les erreurs dans les boucles.

Troisième partie

Implantation et étude expérimentale

Cette partie présente nos expérimentations réalisées pour valider notre approche. Elle compare également les performances de notre algorithme avec celui de `BugAssist`*. Nous commençons par présenter notre implémentation de* `LocFaults` *et l'outil* `BugAssist`*. Par la suite, nous présentons nos résultats d'expérimentations sur notre outil et les résultats de comparaisons avec* `BugAssist`*, ainsi que les discussions.*

Chapitre 5

Expérimentation

"L'inconvénient le plus serieux (et évident) du Model Checking est le problème d'explosion combinatoire. La taille du graphe global d'état peut être (au moins) exponentielle en fonction de la taille du texte du programme." – Allen Ernest Emerson, The Beginning of Model Checking : A Personal Perspective, 2008, P.11

Sommaire

5.1 Introduction

Dans ce chapitre, nous décrivons notre outil (`LocFaults`) de localisation d'erreurs à partir de contre-exemple, et l'outil `BugAssist`. Nous décrivons ensuite les benchmarks et présentons les résultats comparatifs et les discussions.

5.2 Implémentation de LocFaults

Nous avons implémenté notre approche LocFaults. Notre outil est écrit en JAVA. Nous utilisons l'outil de **bounded model checking** CPBPV [Collavizza 2010] pour générer le CFG en forme DSA, et un contre-exemple lorsque le programme en entrée est non-conforme vis-à-vis de sa spécification. Notre outil accepte seulement les programmes avec des calculs sur les entiers. Pour analyser le programme d'entrée, notre outil utilise Eclipse JDT (Java Development Tools). Les noeuds du CFG sont des ensembles de contraintes. Notre outil accepte en entrée un programme JAVA avec une spécification formalisée en JML (Java Modeling Language). Si le programme est non-conforme vis-à-vis de la spécification, l'outil CPBPV produit un contre-exemple, c'est-à-dire une instanciation des variables pour laquelle l'exécution du programme ne satisfait pas la postcondition.

Nous avons utilisé les solveurs de contraintes IBM ILOG MIP [1] et CP [2] de CPLEX, où IBM ILOG MIP est l'outil de programmation linéaire mixte en nombres entiers et IBM ILOG CP est l'outil de programmation par contraintes. Il faut toutefois noter que cette résolution n'est correcte que sur les entiers et que la prise en compte des nombres flottants nécessite l'utilisation d'un solveur spécialisé pour le traitement des flottants. Pour calculer les MCSs, nous avons adapté et implémenté l'algorithme de Liffiton et Sakallah [Liffiton 2008], voir alg. 18.

5.3 L'outil BugAssist

Nous avons comparé LocFaults à l'outil BugAssist [Jose 2011b, Jose 2011d]. BugAssist [3] est un outil qui aide les programmeurs à localiser les bugs en fournissant un ensemble d'erreurs potentielles dans le code. Cet outil prend en entrée un programme ANSI-C annoté avec des assertions. Il utilise un vérificateur de modèle pour trouver les violations des assertions potentielles (contre-exemples). L'algorithme de BugAssist formule le problème de localisation d'erreurs comme un problème MaxSAT et utilise des solveurs partiels MaxSAT.

À partir d'un contre-exemple, BugAssist (voir alg. 19) commence par construire deux formules booléennes : ϕ_H et ϕ_S (lignes 9,10) :

- ϕ_H est la conjonction des trois sous-formules : la formule codant le contre-exemple (nommée $[[test]]$), la formule codant la postcondition (nommée p) et la formule codant la première partie de formule de la trace augmentée par des variables de sélection (nommée TF_1), voir ligne 9 ;
- ϕ_S est la conjonction de toutes les variables de sélection, nommée TF_2 (voir

1. Disponible à l'adresse http ://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/
2. Disponible à l'adresse http ://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/
3. Disponible à l'adresse http://bugassist.mpi-sws.org/.

Algorithm 19: Algorithme de BugAssist

Entrées: Programme P et assertion p

Sorties: Soit p est satisfiable pour toutes les exécutions ou des lignes de bugs potentiels

début
  $(test, \sigma)$ = GenerateCounterexample(P, p) ; % appel à la fonction GenerateCounterexample pour générer les exécutions fautives pour l'assertion p.
  si σ est "Rien" **alors**
    % Aucune exécution fautive n'est trouvée.
    retourner Pas de contre-exemple trouvé pour p ;
  fin
  else
    % Une exécution fautive est trouvée : GenerateCounterexample fournit un contre-exemple ($test$) et une trace de programme qui prouve l'échec de l'assertion (σ).
    $\phi_H = [[test]] \wedge p \wedge TF_1(\sigma)$;
    $\phi_S = TF_2(\sigma)$;
    tant que vrai **faire**
      $BugLoc$ = CoMSS(ϕ_H, ϕ_S) ;
      si BugLoc = $\emptyset$ **alors**
        return Pas plus d'instructions suspects ;
      fin
      else
        Afficher "bug potentiel au CoMSS BugLoc" ;
        $\beta = \vee \{\lambda_i | \lambda_i \in BugLoc\}$;
        $\phi_S = \phi_S \setminus \beta$ et $\phi_H = \phi_H \cup \beta$
      end
    fin
  end
fin

ligne 10).

Il utilise un solver partiel MaxSAT (PMaxSAT). Toutes les clauses dans ϕ_H et ϕ_S sont considérées respectivement comme des clauses dures et molles. La recherche de localisation est réalisée dans la boucle While (lines 11-21). A chaque itération il appelle le solver PMaxSAT pour calculer un CoMSS(ensemble complément de l'ensemble maximal de clauses de la formule de la trace augmentée qui peuvent être simultanément satisfaites), un MCS, en considérant des clauses dures et molles. Toutes les instructions qui correspondent aux clauses du CoMSS généré sont suspectées, elles sont affichées en sortie. Après il bloque l'apparition du CoMSS calculé : il ajoute la clause de blocage correspondant à l'ensemble des clauses dures. Une clause de blocage représente la disjonction des variables de sélection des clauses du CoMSS (voir lignes 18-19). Il réitère la boucle While, jusqu'à ce que le solveur n'arrive pas de trouver un autre CoMSS.

5.4 Expérimentations

Pour évaluer la méthode que nous avons proposée, nous avons comparé les performances de notre outil et de `BugAssist` [Jose 2011b, Jose 2011d] sur un ensemble de programmes. Cette section présente les résultats expérimentaux que nous avons obtenus. Nous commençons par décrire les programmes expérimentés dans la section 5.4.1. Ensuite, nous donnons le protocole expérimental (voir la section 5.4.2). Puis, nous présentons nos premiers résultats (voir la section 5.4.3). Nos résultats de la deuxième série d'expérimentations sont présentés dans la section 5.4.4. Le reste des expérimentations sont donnés dans la section 5.4.5.

5.4.1 Les programmes de l'expérience

Dans cette section, nous allons présenter les programmes erronés que nous avons construits[4]. Nous avons réparti notre base de benchmarks en trois grandes catégories : programmes sans boucles, avec boucles et réalistes. Le but de l'utilisation de ces programmes consiste à valider notre approche de localisation d'erreurs (`LocFaults`) en comparant ses performances avec celles de `BugAssist`.

5.4.1.1 Programmes sans boucles et sans calculs non linéaires

Nous avons d'abord utilisé un ensemble de programmes académiques de petite taille (entre 15 et 100 lignes) sans boucles et sans calculs non linéaires. A savoir :

- **AbsMinus**. Ce programme (voir fig. 4.1) prend en entrée deux entiers i et j et renvoie la valeur absolue de $i-j$. Ci-dessous les versions erronées considérées :

4. Le code source de l'ensemble de programmes est disponible à l'adresse : `http://www.i3s.unice.fr/~bekkouch/Benchs_Mohammed.html`

- AbsMinusKO : L'erreur dans ce programme est sur l'instruction d'affectation "$result = i - j$" (ligne 17), l'instruction correcte devrait être "$result = j - i$". En prenant comme entrée de ce programme $\{i = 0, j = 1\}$, il retourne la valeur -1 comme valeur absolue de $i - j$, alors qu'il devait retourner 1. Ce programme présente le cas où toutes les condition sont correctes.
- AbsMinusKO2 : L'erreur dans ce programme est sur l'instruction d'affectation "$result = i + 1$" (ligne 11), l'instruction correcte devrait être "$result = i$". En prenant le contre-exemple $\{i = 0, j = 1\}$, le programme échoue et renvoie la valeur 0, alors qu'il devait renvoyer la valeur 1 comme valeur absolue de $i - j$. Toutes les conditions dans ce programme sont correctes.
- AbsMinusKO3 : L'erreur dans ce programme est sur l'instruction d'affectation "$k = k+2$" (ligne 14), l'instruction correcte devrait être "$k = k+1$". En prenant l'entrée $\{i = 0, j = 1\}$, cette erreur permet de contredire la condition "$(k == 1 \wedge i! = j)$" et donc l'exécution de l'instruction "$result = i - j$" (ligne 20), ce qui cause une sortie autre que celle attendue, -1 au lieu de 1.
- AbsMinusV2KO, AbsMinusV2KO2 : AbsMinusV2 représente une réécriture du programme qui calcule la valeur absolue de (i - j) (i et j sont les entrées) de telle sorte qu'il devient plus simple : le programme obtenu contient 4 instructions : les lignes 11, 12, 13 et 16. Autrement dit, AbsMinusV2 est sémantiquement équivalent à AbsMinus. Les programmes AbsMinusV2KO et AbsMinusV2KO2 (versions erronées du programme AbsMinusV2) sont équivalents respectivement aux programmes AbsMinusKO et AbsMinusKO2.

- **Minmax**. Ce programme (voir fig. 5.1) prend en entrée trois entiers : $in1$, $in2$ et $in3$, et permet d'affecter la plus petite valeur à la variable *least* et la plus grande valeur à la variable *most*. Ci-dessous la version erronée considérée :
  - MinmaxKO : L'erreur dans ce programme est sur l'instruction d'affectation "$most = in2$" (ligne 19), l'instruction devrait être "$least = in2$". Pour une entrée égale à $\{in1 = 2, in2 = 1, in3 = 3\}$, on devrait avoir comme sortie $least = 1$ et $most = 3$. Avec cette entrée le programme sort $least = 2$ et $most = 1$, ce qui est non-conforme avec la postcondition posée "$(least <= most)$".

```
1  class Minmax {
2      /*@ ensures (least <= most);
3        @*/
4      void minmax(int in1,int in2,int in3){
5      int least; int most;
6      least = in1; most = in1;
7      if (most < in2){
8         most = in2;
9      }
10     if (most < in3){
11        most = in3;
12     }
13     if (least > in2){
14        least = in2;
15     }
16     if (least > in3){
17        least = in3;
18     }}
19 }
```

FIGURE 5.1 – Le programme Minmax.

Mid. Ce programme (voir fig. 2.1) a été utilisé dans le papier [Jones 2002] pour introduire et expliquer l'approche `Tarantula`. Nous avons utilisé la version erronée présente dans le papier mentionné (nommée MidKO).

- **Maxmin6var**. Ce programme (voir fig. 5.2 et fig. 5.3) prend en entrée six variables à valeurs entières et permet d'affecter :
  - la plus grande valeur à la variable max ;
  - et la plus petite valeur à la variable min.

  Nous avons utilisé 4 versions erronées :
  - Maxmin6varKO : L'erreur dans ce programme porte sur l'instruction If $(e > f)$ (ligne 27). Le contre-exemple suivant $\{a = 1, b = -4, c = -3, d = -1, e = 0, f = -4\}$ permet de produire un mauvais branchement et de calculer la sortie suivante $\{max = 1, min = 0\}$ au lieu de $\{max = 1, min = -4\}$.
  - Maxmin6varKO2 : L'erreur dans ce programme est provoquée dans la condition if $((a > b) \wedge (a > c) \wedge (b > d) \wedge (a > e) \wedge (a > f))$ (ligne 12), l'instruction correcte devait être if $((a > b) \wedge (a > c) \wedge (a > d) \wedge (a > e) \wedge (a > f))$. Cette erreur crée un mauvais branchement, en prenant comme entrée le contre-exemple suivant $\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$; elle entraîne la sortie suivante $\{max = 0, min = -3\}$ au lieu de $\{max = 1, min = -3\}$.
  - Maxmin6varKO3 : Les erreurs dans ce programme sont introduites au niveau des deux conditions (lignes 12 et 15). En prenant comme entrée $\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$, ces erreurs produisent deux mauvais branchements ; le programme renvoie $\{max = 0, min = -3\}$ au lieu de $\{max = 1, min = -3\}$.
  - Maxmin6varKO4 : Ce programme contient 3 erreurs conditionnelles : lignes 12, 15 et 20. En prenant l'entrée suivante $\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$, le programme entraîne la sortie suivante $\{max = -1, min = -4\}$ au lieu de $\{max = 1, min = -4\}$.

```
/*
 * Find the maximum and minimum of six values.
 */
class Foo{

    /*@ requires ( (a==1) && (b==-4) && (c==-3) && (d==-1) && (e==0) && (f==-4) );
      @ ensures ( (max >= a) && (max >= b) && (max >= c) && (max >= d) && (max >= e) && (
           max >= f) && (min <= a) && (min <= b) && (min <= c) && (min <= d) && (min <= e)
            && (min <= f) );
      @*/
    static void maxmin (int a, int b, int c, int d, int e, int f) {
  int max;
        int min;
  if ((a>b) && (a>c) && (a>d) && (a>e) && (a>f)){
            max=a;

            if ((b<c) && (b<d) && (b<e) && (b<f)){
              min=b;
            }
            else{
                if ((c<d) && (c<e) && (c<f)){
                   min=c;
                }
                else{
                     if ((d<e) && (d<f)){
                        min=d;
                     }
                     else{
                         if (e<f){
                              min=e;
                         }
                         else{
                              min=f;
                         }
                     }
                }
            }

  }
    else{
      if ((b>c) && (b>d) && (b>e) && (b>f)){
        max=b;

               if ((a<c) && (a<d) && (a<e) && (a<f)){
                  min=a;
                }else{
                     if ((c<d) && (c<e) && (c<f)){
                       min=c;
                     }
                     else{
                         if ((d<e) && (d<f)){
                             min=d;
                         }
                         else{
                             if (e<f){
                                min=e;
                             }
                             else{
                                min=f;
                             }
                         }
                     }
                }

            }
      else{
        if ((c>d) && (c>e) && (c>f)){
             max=c;

                    if ((a<b) && (a<d) && (a<e) && (a<f)){
                       min=a;
                    }else{
                         if ((b<d) && (b<e) && (b<f)){
                           min=b;
                         }
                         else{
                             if ((d<e) && (d<f)){
                               min=d;
                             }
                             else{
                                 if (e<f){
                                   min=e;
                                 }
                                 else{
                                   min=f;
                                 }
                             }
                         }
                    }

    }
    else{
```

FIGURE 5.2 – Portion du programme Maxmin6var.

```
1           if ((d>e) && (d>f)){
2               max=d;
3
4                               if ((a<b) && (a<c) && (a<e) && (a<f)){
5                                   min=a;
6                               }else{
7                                    if ((b<c) && (b<e) && (b<f)){
8                                      min=b;
9                                    }
10                                   else{
11                                       if ((c<e) && (c<f)){
12                                          min=c;
13                                       }
14                                       else{
15                                           if (e<f){
16                                             min=e;
17                                           }
18                                           else{
19                                             min=f;
20                                           }
21                                       }
22                                   }
23                               }
24          }
25          else{
26           if (e>f){
27             max=e;
28
29                                    if ((a<b) && (a<c) && (a<d) && (a<f)){
30                                       min=a;
31                                    }else{
32                                         if ((b<c) && (b<d) && (b<f)){
33                                            min=b;
34                                         }
35                                         else{
36                                             if ((c<d) && (c<f)){
37                                                min=c;
38                                             }
39                                             else{
40                                                 if (d<f){
41                                                    min=d;
42                                                 }
43                                                 else{
44                                                    min=f;
45                                                 }
46                                             }
47                                         }
48                                     }
49          }
50          else{
51               max=f;
52
53                                    if ((a<b) && (a<c) && (a<d) && (a<e)){
54                                       min=a;
55                                    }else{
56                                         if ((b<c) && (b<d) && (b<e)){
57                                            min=b;
58                                         }
59                                         else{
60                                             if ((c<d) && (c<e)){
61                                                min=c;
62                                             }
63                                             else{
64                                                 if (d<e){
65                                                    min=d;
66                                                 }else{
67                                                    min=e;
68                                                 }
69                                             }
70                                        }
71                                    }
72                }
73          }
74          }
75          }
76          }
77          }
78  }
```

FIGURE 5.3 – Le suite du programme Maxmin6var.

- **Tritype**. Ce programme (voir fig. 5.4) est classique, il a été utilisé très souvent en test et vérification de programmes. Il prend en entrée trois entiers (les côtés d'un triangle) et retourne 3 si les entrées correspondent à un triangle équilatéral, 2 si elles correspondent à un triangle isocèle, 1 si elles correspondent à un autre type de triangle, 4 si elles ne correspondent pas à un triangle valide. Ci-dessous les versions erronées considérées :
  - TritypeKO : L'erreur dans ce programme est sur l'instruction d'affectation "$trityp = 1$" (ligne 54), l'instruction correcte devrait être "$trityp = 2$". En prenant comme entrée les valeurs $\{i = 2, j = 3, k = 2\}$, le programme exécute l'instruction erronée et renvoie 1, alors qu'il devait renvoyer 2 (le triangle en entrée est un isocèle). Le programme ne présente aucune condition incorrecte.
  - TritypeKO2 : L'erreur dans ce programme est sur la condition "$(trityp == 1 \wedge (i+k) > j)$" (ligne 53), l'instruction devrait être "$(trityp == 2 \wedge (i+k) > j)$". En prenant comme entrée $\{i = 2, j = 2, k = 2\}$, le programme renvoie 2 pour indiquer que le triangle est un isocèle, alors qu'il devait renvoyer la valeur 4 pour signaler que l'entrée ne correspond pas à un triangle valide.
  - TritypeKO2V2 : L'erreur dans ce programme est sur l'instruction d'affectation de la ligne 31 : "$trityp = trityp + 1$", l'instruction correcte devrait être "$trityp = trityp + 2$". En prenant comme contre-exemple l'entrée $\{i = 1, j = 2, k = 1\}$, le programme renvoie la valeur 2 (l'entrée correspond à un isocèle), alors qu'il devait renvoyer la valeur 4 (l'entrée ne correspond pas à un triangle valide).
  - TritypeKO3 : L'erreur dans ce programme est sur l'instruction conditionnelle "$(trityp == 2 \wedge (i + j) > k)$" (ligne 53), l'instruction correcte est "$(trityp == 2 \wedge (i + k) > j)$". Avec l'entrée $\{i = 1, j = 2, k = 1\}$, le programme devrait renvoyer que le triangle en entrée est un isocèle (sortie égale à 2). Ce programme avec cette même entrée renvoie que le triangle correspondant n'est pas valide.
  - TritypeKO4 : L'erreur dans ce programme est sur l'instruction "$(trityp >= 3)$", l'instruction devrait être "$(trityp > 3)$". En prenant l'entrée suivante $\{i = 2, j = 3, k = 3\}$, le programme satisfait la condition et exécute l'instruction "$trityp = 3$" (ligne 46) pour renvoyer que le triangle en entrée est un équilatéral, alors qu'il devrait la contredire et renvoyer que qu'il est isocèle (la valeur renvoyée devrait être égale à 2).
  - TritypeKO5 : Les erreurs dans ce programme sont sur la condition "$(j! = k)$" (ligne 32) et sur la condition "$(trityp >= 3)$" (ligne 45), les instructions correctes devront être respectivement "$(j == k)$" et "$(trityp >= 3)$". En prenant l'entrée $\{i = 2, j = 3, k = 3\}$, le programme renvoie la valeur 1, alors qu'il devait renvoyer la valeur 2 (le triangle est un isocèle).
  - TritypeKO6 : Les erreurs dans ce programme sont sur l'instruction conditionnelle "$(j! = k)$" (ligne 32) et sur l'affectation "$trityp = trityp + 4$" (ligne 33), les instructions correctes devront être respectivement "$(j == k)$" et "$trityp = trityp + 3$". En prenant comme entrée du programme

$\{i = 2, j = 3, k = 3\}$, il retourne que le triangle est équilatéral (la valeur 1), or il devait renvoyer qu'il est isocèle (la valeur 2).

```
1  /* program for triangle classification
2   * returns 1 if (i,j,k) are the sides of any triangle
3   * returns 2 if (i,j,k) are the sides of an isosceles triangle
4   * returns 3 if (i,j,k) are the sides of an equilateral triangle
5   * returns 4 if (i,j,k) are not the sides of a triangle
6   */
7  class Tritype {
8    /*@ requires
9      @ (i >= 0 && j >= 0 && k >= 0);
10     @ ensures
11     @ (((i+j) <= k || (j+k) <= i || (i+k) <= j) ==> (\result == 4))
12     @  && ((!((i+j) <= k || (j+k) <= i || (i+k) <= j) && (i==j && j==k)) ==> (\result ==
           3))
13     @  && ((!((i+j) <= k || (j+k) <= i || (i+k) <= j) && !(i==j && j==k) && (i==j || j==k
           || i==k)) ==> (\result == 2))
14     @  && ((!((i+j) <= k || (j+k) <= i || (i+k) <= j) && !(i==j && j==k) && !(i==j || j==
           k || i==k)) ==> (\result == 1));
15     @*/
16   static int caller (int i, int j, int k) {
17     int trityp;
18     if (i == 0 || j == 0 || k == 0) {
19       trityp = 4;
20     }
21     else {
22       trityp = 0;
23       if (i == j) {
24         trityp = trityp + 1;
25       }
26       if (i == k) {
27         trityp = trityp + 2;
28       }
29       if (j == k) {
30         trityp = trityp + 3;
31       }
32       if (trityp == 0) {
33         if ((i+j) <= k || (j+k) <= i || (i+k) <= j) {
34           trityp = 4;
35         }
36         else {
37           trityp = 1;
38         }
39       }
40       else {
41         if (trityp > 3) {
42           trityp = 3;
43         }
44         else {
45           if (trityp == 1 && (i+j) > k) {
46             trityp = 2;
47           }
48           else {
49             if (trityp == 2 && (i+k) > j) {
50               trityp = 2;
51             }
52             else {
53               if (trityp == 3 && (j+k) > i) {
54                 trityp = 2;
55               }
56               else {
57                 trityp = 4;
58               }
59             }
60           }
61         }
62       }
63     }
64     return trityp;
65   }
66 }
```

FIGURE 5.4 – Le programme Tritype.

– **TriPerimetre**. Ce programme (voir fig. 5.5) a exactement la même structure de contrôle que tritype. La différence est que TriPerimetre renvoie la somme

des côtés du triangle si les entrées correspondent à un triangle valide, et -1 dans le cas inverse. Ci-dessous les versions erronées considérées :

- TriPerimetreKO : L'erreur dans ce programme est sur l'instruction d'affectation "$res = 2*i+k$" (ligne 58), l'instruction devrait être "$res = 2*i+j$". En prenant l'entrée $\{i = 2, j = 1, k = 2\}$, le programme renvoie comme somme des cotés du triangle correspondant la valeur 6, alors qu'il devait renvoyer plutôt la valeur 5. Toutes les conditions dans les programme sont correctes.
- TriPerimetreKOV2 : L'erreur dans ce programme est sur l'instruction d'affectation "$res = 2 * j$" (ligne 34), l'instruction correcte devrait être "$res = 2*i$". En prenant comme entrée $\{i = 2, j = 3, k = 2\}$, le programme renvoie que la somme du triangle est 9, alors qu'il devait renvoyer la somme 7. Toutes les conditions présentes dans ce programme sont correctes. Ce programme est sémantiquement équivalent à TriPerimetreKO, la différence est que l'erreur provoquée n'est pas sur la même instruction pour les deux programmes.
- TriPerimetreKO2 : L'erreur dans ce programme est sur la condition "$(trityp == 1 \wedge (i+k) > j)$", l'instruction correcte devrait être "$(trityp == 2 \wedge (i + k) > j)$". En prenant comme entrée $\{i = 1, j = 1, k = 2\}$, le programme renvoie que la somme du triangle correspondant est 4, or il devait renvoyer -1 (les entrées ne correspondent pas à un triangle valide).
- TriPerimetreKO2V2 : L'erreur dans ce programme se situe sur l'affectation "$trityp = trityp + 1$", l'instruction correcte devrait être "$trityp = trityp + 2$". Avec l'entrée suivante $\{i = 1, j = 2, k = 1\}$, le programme calcule la somme du triangle : 4, or il devait renvoyer la valeur -1 comme quoi les entrées ne correspondent pas à un triangle valide.
- TriPerimetreKO3 : L'erreur dans ce programme est sur la deuxième partie de la condition "$(trityp == 1 \wedge (i+j) > k)$" (ligne 57), l'instruction correcte devrait être "$(trityp == 2 \wedge (i + k) > j)$". En prenant l'entrée $\{i = 1, j = 2, k = 1\}$, le programme fournit en sortie la somme 4, or il devrait retourner la valeur -1 pour signaler que les entrées ne correspondent pas à un triangle valide.
- TriPerimetreKO4 : L'erreur dans ce programme se trouve sur l'instruction conditionnelle "$(trityp >= 3)$", l'instruction devrait être "$(trityp > 3)$". Avec l'entrée suivante $\{i = 2, j = 3, k = 3\}$, le programme exécute l'instruction "$res = 3 * i$" et renvoie la valeur 6, or il devait renvoyer la valeur 8 (la somme des cotés du triangle en entrée du programme).
- TriPerimetreKO5 : Dans ce programme, nous avons introduit deux erreurs : sur la condition "$(j! = k)$" (ligne 34) et sur la condition "$(trityp >= 3)$" (ligne 49), les instructions correctes devront être respectivement "$(j == k)$" et "$(trityp >= 3)$". Avec l'entrée suivante $\{i = 2, j = 2, k = 3\}$, ce programme doit renvoyer la valeur 7.
- TriPerimetreKO6 : Les erreurs présentes dans ce programme se situent sur la condition "$(j! = k)$" (ligne 34) et sur l'affectation "$trityp = trityp+4$" (ligne

35), les instructions correctes devront être respectivement "$(j == k)$" et "$trityp = trityp + 3$". Avec le contre-exemple suivant $\{i = 2, j = 2, k = 3\}$, TriPerimetreKO6 devait retourner la somme 7.

```
1  /* program for triangle perimeter
2   * returns i+j+k
3   *       i+j+k if (i,j,k) are the sides of any triangle
4   *       2*i + j or 2*i+k or 2*j+i if (i,j,k) are the sides of an isosceles triangle
5   *       3*i if (i,j,k) are the sides of an equilateral triangle
6   * returns -1 if (i,j,k) are not the sides of a triangle
7   *
8   */
9
10 class TriPerimetreKO{
11   /*@ requires
12     @ (i >= 0 && j >= 0 && k >= 0);
13     @ ensures
14     @ (((i+j) <= k || (j+k) <= i || (i+k) <= j) ==> (\result == -1))
15     @  && ((!((i+j) <= k || (j+k) <= i || (i+k) <= j) && (i==j && j==k)) ==> (\result ==
         i+j+k));
16     @*/
17   static int caller (int i, int j, int k) {
18     int trityp = 0;
19     int res;
20     if (i == 0 || j == 0 || k == 0) {
21       trityp = 4;
22       res = -1;
23     }
24     else {
25       trityp = 0;
26       if (i == j) {
27         trityp = trityp + 1;
28       }
29       if (i == k) {
30         trityp = trityp + 2;
31       }
32       if (j == k) {
33         trityp = trityp + 3;
34       }
35       if (trityp == 0) {
36         if ((i+j) <= k || ((j+k) <= i || (i+k) <= j)) {
37           trityp = 4;
38           res = -1;
39         }
40         else {
41           trityp = 1;
42           res = i+j+k;
43         }
44       }
45       else {
46         if (trityp > 3) {
47           res = 3*i;
48         }
49         else {
50           if (trityp == 1 && (i+j) > k) { // i==j
51             res=2*i+k;
52           }
53           else {
54             if (trityp == 2 && (i+k) > j) {    // i==k
55               res = 2*i+j;
56             }
57             else {
58               if (trityp == 3 && (j+k) > i) {   // j==k
59                 res=2*j+i;
60               }
61               else {
62                 res=-1;
63               }
64             }
65           }
66         }
67       }
68     }
69     return res;
70   }
71 }
```

FIGURE 5.5 – Le programme TriPerimetre.

5.4.1.2 Programmes sans boucles et avec calculs non linéaires

Nous avons aussi utilisé un ensemble de programmes sans boucles et avec calculs non linéaires :

- **TriMultPerimetre**. Ce programme (voir fig. 5.6) a aussi la même structure de contrôle que tritype. Mais en plus du type de triangle, il calcule le produit des cotés et retourne ce produit s'il s'agit d'un triangle (sinon il retourne -1). Voici les versions erronées utilisées :
  - TriMultPerimetreKO : L'erreur provoquée dans le programme est sur la ligne 58, l'instruction devrait être "$res = i*i*j$" au lieu de "$res = i*i*k$".
  - TriMultPerimetreKO2 : L'erreur injectée dans ce programme est sur la condition de ligne 57 : "$(trityp == 1 \wedge (i + k) > j)$", l'instruction devrait être "$(trityp == 2 \wedge (i + k) > j)$".
  - TriMultPerimetreKO2V2 : L'erreur insérée dans ce programme est sur l'affectation "$trityp = trityp+1$", sur la ligne 33 ; l'instruction correcte devrait être "$trityp = trityp + 2$".
  - TriMultPerimetreKO3 : L'erreur affectant ce programme se trouve à la ligne 56 (erreur conditionelle) : "$(i + j) > k$" au lieu de "$(i + k) > j$".
  - TriMultPerimetreKO4 : Le problème dans ce programme vient de la ligne 48 : la condition "$(trityp >= 3)$" est erronée, l'instruction correcte est "$(trityp > 3)$".
  - TriMultPerimetreKO5 : Ce programme a deux instructions erronées, la première est sur la ligne 33 : la condition "$(j! = k)$", et la deuxième est au niveau de la ligne 48 : la condition "$(trityp >= 3)$" ; pour restaurer la conformité vis-à-vis de la postcondition, il faut les remplacer respectivement par "$(j == k)$" et "$(trityp > 3)$".
  - TriMultPerimetreKO6 : Ce programme contient une erreur à la ligne 33 (la condition "$(j! = k)$") et une autre à la ligne suivante (l'affectation "$trityp = trityp + 4$") ; ces instructions devrait être respectivement "$(j == k)$" et "$trityp = trityp + 4$".

```
1  /* program for triangle perimeter
2   * returns i*j*k
3   *       i*j*k if (i,j,k) are the sides of any triangle
4   *       i*i * j or 2*i+k or 2*j+i if (i,j,k) are the sides of an isosceles triangle
5   *       i*i*i if (i,j,k) are the sides of an equilateral triangle
6   * returns -1 if (i,j,k) are not the sides of a triangle
7   */
8
9  class TriMultPerimetreKO{
10   /*@ requires
11     @ (i >= 0 && j >= 0 && k >= 0);
12     @ ensures
13     @ (((i+j) <= k || (j+k) <= i || (i+k) <= j) ==> (\result == -1))
14     @  && ((!((i+j) <= k || (j+k) <= i || (i+k) <= j) && (i==j && j==k)) ==> (\result ==
           i*j*k));
15     @*/
16   static int caller (int i, int j, int k) {
17     int trityp = 0;
18     int res;
19     if (i == 0 || j == 0 || k == 0) {
20       trityp = 4;
21       res = -1;
22     }
23     else {
24       trityp = 0;
25       if (i == j) {
26         trityp = trityp + 1;
27       }
28       if (i == k) {
29         trityp = trityp + 2;
30       }
31       if (j == k) {
32         trityp = trityp + 3;
33       }
34       if (trityp == 0) {
35         if ((i+j) <= k || ((j+k) <= i || (i+k) <= j)) {
36           trityp = 4;
37           res = -1;
38         }
39         else {
40           trityp = 1;
41           res = i*j*k;
42         }
43       }
44       else {
45         if (trityp > 3) {
46           res = i*i*i;
47         }
48         else {
49           if (trityp == 1 && (i+j) > k) { // i==j
50             res=i*i*k;
51           }
52           else {
53             if (trityp == 2 && (i+k) > j) {     // i==k
54               res = i*i*j;
55             }
56             else {
57               if (trityp == 3 && (j+k) > i) {   // j==k
58                 res=j*j*i;
59               }
60               else {
61                 res=-1;
62               }
63             }
64           }
65         }
66       }
67     }
68     return res;
69   }
70 }
```

FIGURE 5.6 – Le programme TriMultPerimetre.

- **Heron**. Ce programme (voir fig. 5.7) a aussi la même structure de contrôle que tritype mais avec des expressions non-linéaires. Avec la différence qu'il renvoie le carré de l'aire du triangle (i,j,k) en entrée en utilisant la formule de Héron. Voici les versions erronées utilisées :
  - HeronKO : Ce programme présente le même type d'erreur que dans TritypeKO et TriPerimetreKO (l'erreur est sur un calcul final dans le chemin du contre-exemple), l'instruction mutée est sur la ligne 61 : "$res = s * (s - i) * (s - j) * (s - i)$" au lieu de "$res = s * (s - i) * (s - j) * (s - j)$".
  - HeronKO2 : L'erreur dans ce programme est injectée sur la condition de la ligne 59 (même type d'erreur que dans TritypeKO2 et TriPerimetreKO2) : $trityp == 1$ au lieu de $trityp == 2$. Le contre-exemple associé à ce programme est le suivant : $\{i = 2, j = 2, k = 4\}$. En prenant en entrée ce cas d'erreur, le programme HeronKO2 calcule la valeur 32 en exécutant l'instruction "$res = s * (s - i) * (s - j) * (s - i)$" de la ligne 62 ; or il devrait retourner la valeur -1 comme quoi les entrées du programme correspondent à un triangle invalide.
  - HeronV1, HeronV2 : Les programmes HeronV1 et HeronV2 sont équivalents respectivement aux programmes HeronKO et HeronKO2. La différence est que dans HeronV1 et HeronV2, l'instruction "$s = (i + j + k)/2$" (ligne 19) est supprimée ; l'expression "$(i + j + k)/2$" est substituée à "s" dans les instructions qui calculent "res".
  - HeronKO2V2 : L'erreur dans HeronKO2V2 est sur la ligne 31 : l'affectation "$trityp = trityp + 1$" ; pour la corriger, il faut mettre à sa place "$trityp = trityp + 2$".
  - HeronKO3 : L'erreur dans HeronKO3 est située sur la condition de la ligne 59 : "$(i + j) > k$" au lieu de "$(i + k) > j$".
  - HeronKO4 : L'erreur dans HeronKO4 est sur la condition de la ligne 47 : "$trityp >= 3$" au lieu de "$trityp > 3$". Avec l'entrée suivante $\{i = 2, j = 3, k = 3\}$, le programme devrait calculer la valeur 8.
  - HeronKO5 : Il y a deux erreurs conditionelles dans HeronKO5 sur les lignes 32 et 47 : "$(j! = k)$" et "$(trityp >= 3)$" au lieu de respectivement "$(j == k)$" et "$(trityp > 3)$".
  - HeronKO6 : Ce programme contient deux erreurs, une conditionelle (ligne 32 : "$(j! = k)$") et une affectation (ligne 33 : "$trityp = trityp + 4$") ; ces instructions devrait être respectivement "$(j == k)$" et "$trityp = trityp+3$".

```
/* program that computes the square of the area of a triangle.
It uses the Heron formula as specification. It uses a specific formula to compute the
     area when the triangle is equilateral and uses some variations of the Heron formula
     when the triangle is isosceles.
We assume that the sum of the sides is even to guarantee that the expression (i+j+k)/2
     used in Heron formula is an integer.

This example illustrates the case of a program with numerical computations
 */

class HeronKO {
  /*@ requires
    @ ((i >= 0 && j >= 0 && k >= 0) && (2*(i+j+k)/2==(i+j+k));
    @ ensures
    @ (((((i+j) <= k || (j+k) <= i || (i+k) <= j) || (i==0 && j==0 && k==0)) ==> (\result
          == -1))
    @  && (!(((i+j) <= k || (j+k) <= i || (i+k) <= j) || (i==0 && j==0 && k==0))) ==> (\
        result == (i+j+k)/2*((i+j+k)/2-i)*((i+j+k)/2-j)*((i+j+k)/2-k)));
    @*/
  static int caller (int i, int j, int k) {
    int trityp = 0;
    int res = 0;
    int s = (i+j+k)/2;
    if (i == 0 || j == 0 || k == 0) {
      trityp = 4;
      res = 0;
    }
    else {
      trityp = 0;
      if (i == j) {
        trityp = trityp + 1;
      }
      if (i == k) {
        trityp = trityp + 2;
      }
      if (j == k) {
        trityp = trityp + 3;
      }
      if (trityp == 0) {
        if ((i+j) <= k || ((j+k) <= i || (i+k) <= j)) {
          trityp = 4;
          res = -1;
        }
        else {
            trityp = 1; // any triangle
            res = s*(s-i)*(s-j)*(s-k);
        }
      }
      else {
          if (trityp > 3) { // equilateral
        trityp = 3;
        res = (3*i*i*i*i)/16;
        }
        else {
            if (trityp == 1 && (i+j) > k) { // isosceles
            trityp = 2;
            // i==j
            res = s*(s-i)*(s-i)*(s-k);

          }
          else {
            if (trityp == 2 && (i+k) > j) {
              trityp = 2;
              // i==k
                    res = s*(s-i)*(s-j)*(s-i);
            }
            else {
              if (trityp == 3 && (j+k) > i) {
                trityp = 2;
                // j==k
                res = s*(s-i)*(s-j)*(s-j);
              }
              else {
                trityp = 4;
                res = -1;

              }
            }
          }
        }
      }
    }
    return res;
  }

}
```

FIGURE 5.7 – Le programme Heron

5.4.1.3 Programmes avec boucles

Nous avons utilisé trois programmes avec boucles : BubbleSort, Sum et SquareRoot. Ces programmes contiennent des boucles pour étudier le comportement de notre approche par rapport à `BugAssist` sur ce type de programmes. Pour augmenter la complexité d'un programme, nous augmentons le nombre d'itérations dans les boucles à l'exécution de chaque outil ; nous utilisons la même borne de dépliage des boucles pour lancer `LocFaults` et `BugAssist`.

- BubbleSort (voir fig. 5.8) est une implémentation de l'algorithme de tri à bulles. Ce programme contient deux boucles imbriquées ; sa complexité en moyenne est d'ordre n^2, où n est la taille du tableau : le tri à bulles est considéré parmi les mauvais algorithmes de tri. L'instruction erronée dans ce programme entraîne le programme à trier le tableau en entrée en considérant seulement ses $n-1$ premiers éléments. Le mauvais fonctionnement du BubbleSort est dû au nombre d'itérations insuffisant effectué par la boucle. Cela est dû à l'initialisation fautive de la variable j : j = tab.length - 1 ; l'instruction devrait être j = tab.length.

```
1  class BubbleSort {
2      /*@ ensures
3        @ (\forall int k2;
4        @    (k2 >= 0 && k2 < tab.length - 1);
5        @           tab[k2] <= tab[k2 + 1]);
6        @*/
7
8      void bubbleSort (int[] tab) {
9        int i = 0;
10       int j = tab.length - 1; /*error : the instruction should be j = tab.length */
11       int aux = 0;
12       int fini = 0;
13       while (fini == 0) {
14            fini = 1;
15            i = 1;
16            while (i < j) {
17
18              if (tab[i-1] > tab[i]) {
19                 aux = tab[i-1];
20                 tab[i-1] = tab[i];
21                 tab[i] = aux;
22                 fini = 0;
23              }
24              i = i + 1;
25            }
26            j = j - 1;
27       }
28       return;
29      }
30 }
```

FIGURE 5.8 – Le programme BubbleSort

- Le programme SquareRoot (voir fig. 5.9) permet de trouver la partie entière de la racine carrée du nombre entier 50. Une erreur est injectée à la ligne 13, qui entraîne à retourner la valeur 8 ; or le programme doit retourner 7. Ce programme a été utilisé dans le papier décrivant l'approche `BugAssist`, il contient un calcul numérique linéaire dans sa boucle et non linéaire dans sa postcondition.

```
class SquareRoot{
  /*@ ensures((res*res<=val) && (res+1)*(res+1)>val);*/
  int SquareRoot()
    {
        int val = 50;
        int i = 1;
        int v = 0;
        int res = 0;
        while (v < val){
             v = v + 2*i + 1;
             i= i + 1;
        }
        res = i; /*error: the instruction should be res = i - 1*/
        return res;
   }
}
```

FIGURE 5.9 – Le programme SquareRoot

– Le programme Sum (voir fig. 5.10) prend un entier positif n de l'utilisateur, et permet de calculer la valeur $\sum_{i=1}^{n} i$. La postcondition spécifie cette somme. L'erreur dans Sum est dans la condition de sa boucle. Elle cause le calcul de la somme $\sum_{i=1}^{n-1} i$ au lieu de $\sum_{i=1}^{n} i$. Ce programme contient des instructions numériques linéaires dans le cœur de la boucle, et une postcondition non linéaire.

```
class Sum {
   /*@ ensures (\result == (n*(n+1))/2);@*/
  static int Sum (int n) {
    int s = 0;
    int i = 0;
    while (i < n) {
      s = s + i;
      i = i + 1;
    }
    return s;
  }
}
```

FIGURE 5.10 – Le programme Sum

Le code source de l'ensemble des programmes est disponible à l'adresse `http://www.i3s.unice.fr/~bekkouch/Bench_Mohammed.html`.

5.4.1.4 Benchmarks réalistes

Nous avons aussi évalué notre approche sur les programmes TCAS (Traffic Collision Avoidance System) de la suite de test Siemens[Rosenblum 1997] (voir le programme original dans fig. 5.11 et fig. 5.12). Il s'agit là aussi d'un benchmark bien connu qui correspond à un système d'alerte de trafic et d'évitement de collisions aériennes. Il y a 41 versions erronées et 1608 cas de tests. Nous avons utilisé toutes les versions erronées sauf celles dont l'indice $AltLayerValue$ déborde du tableau $PositiveRAAltThresh$ car les débordements de tableau ne sont pas traités dans `CPBPV`. A savoir, les versions **TcasKO...TcasKO41**. Les erreurs dans ces programmes sont provoquées à des endroits différents. 1608 cas de tests sont proposés, chacun correspondant à un contre-exemple. Pour chacun de ces cas de test T_j, on construit un programme $TcasV_iT_j$ qui prend comme entrée le contre-exemple, et

dont la postcondition correspond à la sortie correcte attendue.

```
1
2  /*  -*- Last-Edit:  Fri Jan 29 11:13:27 1993 by Tarak S. Goradia; -*- */
3  /* $Log: tcas.c,v $
4   * Revision 1.2  1993/03/12  19:29:50  foster
5   * Correct logic bug which didn't allow output of 2 - hf
6   * */
7
8  #include <stdio.h>
9
10 #define OLEV        600     /* in feets/minute */
11 #define MAXALTDIFF 600     /* max altitude difference in feet */
12 #define MINSEP      300           /* min separation in feet */
13 #define NOZCROSS    100     /* in feet */
14         /* variables */
15 typedef int bool;
16 int Cur_Vertical_Sep;
17 bool High_Confidence;
18 bool Two_of_Three_Reports_Valid;
19 int Own_Tracked_Alt;
20 int Own_Tracked_Alt_Rate;
21 int Other_Tracked_Alt;
22 int Alt_Layer_Value;     /* 0, 1, 2, 3 */
23 int Positive_RA_Alt_Thresh[4];
24 int Up_Separation;
25 int Down_Separation;
26         /* state variables */
27 int Other_RAC;       /* NO_INTENT, DO_NOT_CLIMB, DO_NOT_DESCEND */
28 #define NO_INTENT 0
29 #define DO_NOT_CLIMB 1
30 #define DO_NOT_DESCEND 2
31 int Other_Capability;    /* TCAS_TA, OTHER */
32 #define TCAS_TA 1
33 #define OTHER 2
34 int Climb_Inhibit;     /* true/false */
35 #define UNRESOLVED 0
36 #define UPWARD_RA 1
37 #define DOWNWARD_RA 2
38
39 void initialize()
40 {
41     Positive_RA_Alt_Thresh[0] = 400;
42     Positive_RA_Alt_Thresh[1] = 500;
43     Positive_RA_Alt_Thresh[2] = 640;
44     Positive_RA_Alt_Thresh[3] = 740;
45 }
46
47 int ALIM ()
48 {
49     return Positive_RA_Alt_Thresh[Alt_Layer_Value];
50 }
51
52 int Inhibit_Biased_Climb ()
53 {
54     return (Climb_Inhibit ? Up_Separation + NOZCROSS : Up_Separation);
55 }
56
57 bool Non_Crossing_Biased_Climb()
58 {
59     int upward_preferred;
60     int upward_crossing_situation;
61     bool result;
62     upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
63     if (upward_preferred)
64     {
65   result = !(Own_Below_Threat()) || ((Own_Below_Threat()) && (!(Down_Separation >= ALIM()
         )));
66     }
67     else
68     {
69   result = Own_Above_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Up_Separation >= ALIM()
         );
70     }
71     return result;
72 }
```

FIGURE 5.11 – Portion du programme Tcas

```
1
2  bool Non_Crossing_Biased_Descend()
3  {
4      int upward_preferred;
5      int upward_crossing_situation;
6      bool result;
7      upward_preferred = Inhibit_Biased_Climb() > Down_Separation;
8      if (upward_preferred)
9      {
10   result = Own_Below_Threat() && (Cur_Vertical_Sep >= MINSEP) && (Down_Separation >= ALIM
         ());
11     }
12     else
13     {
14   result = !(Own_Above_Threat()) || ((Own_Above_Threat()) && (Up_Separation >= ALIM()));
15     }
16     return result;
17 }
18
19 bool Own_Below_Threat()
20 {
21     return (Own_Tracked_Alt < Other_Tracked_Alt);
22 }
23
24 bool Own_Above_Threat()
25 {
26     return (Other_Tracked_Alt < Own_Tracked_Alt);
27 }
28
29 int alt_sep_test()
30 {
31     bool enabled, tcas_equipped, intent_not_known;
32     bool need_upward_RA, need_downward_RA;
33     int alt_sep;
34     enabled = High_Confidence && (Own_Tracked_Alt_Rate <= OLEV) && (Cur_Vertical_Sep >
           MAXALTDIFF);
35     tcas_equipped = Other_Capability == TCAS_TA;
36     intent_not_known = Two_of_Three_Reports_Valid && Other_RAC == NO_INTENT;
37     alt_sep = UNRESOLVED;
38     if (enabled && ((tcas_equipped && intent_not_known) || !tcas_equipped))
39     {
40   need_upward_RA = Non_Crossing_Biased_Climb() && Own_Below_Threat();
41   need_downward_RA = Non_Crossing_Biased_Descend() && Own_Above_Threat();
42   if (need_upward_RA && need_downward_RA)
43         /* unreachable: requires Own_Below_Threat and Own_Above_Threat
44            to both be true - that requires Own_Tracked_Alt < Other_Tracked_Alt
45            and Other_Tracked_Alt < Own_Tracked_Alt, which isn't possible */
46       alt_sep = UNRESOLVED;
47   else if (need_upward_RA)
48       alt_sep = UPWARD_RA;
49   else if (need_downward_RA)
50       alt_sep = DOWNWARD_RA;
51   else
52       alt_sep = UNRESOLVED;
53     }
54     return alt_sep;
55 }
56
57 main(argc, argv)
58 int argc;
59 char *argv[];
60 {
61     if(argc < 13)
62     {
63   fprintf(stdout, "Error: Command line arguments are\n");
64   fprintf(stdout, "Cur_Vertical_Sep, High_Confidence, Two_of_Three_Reports_Valid\n");
65   fprintf(stdout, "Own_Tracked_Alt, Own_Tracked_Alt_Rate, Other_Tracked_Alt\n");
66   fprintf(stdout, "Alt_Layer_Value, Up_Separation, Down_Separation\n");
67   fprintf(stdout, "Other_RAC, Other_Capability, Climb_Inhibit\n");
68   exit(1);
69     }
70     initialize();
71     Cur_Vertical_Sep = atoi(argv[1]);
72     High_Confidence = atoi(argv[2]);
73     Two_of_Three_Reports_Valid = atoi(argv[3]);
74     Own_Tracked_Alt = atoi(argv[4]);
75     Own_Tracked_Alt_Rate = atoi(argv[5]);
76     Other_Tracked_Alt = atoi(argv[6]);
77     Alt_Layer_Value = atoi(argv[7]);
78     Up_Separation = atoi(argv[8]);
79     Down_Separation = atoi(argv[9]);
80     Other_RAC = atoi(argv[10]);
81     Other_Capability = atoi(argv[11]);
82     Climb_Inhibit = atoi(argv[12]);
83     fprintf(stdout, "%d\n", alt_sep_test());
84     exit(0);
85 }
```

FIGURE 5.12 – Suite du programme Tcas

5.4.2 Protocole expérimental

Comme `LocFaults` est basé sur `CPBPV`[Collavizza 2010] qui travaille sur des programmes Java et que `BugAssist` travaille sur des programmes C, nous avons construit pour chacun des programmes :

- une version en Java annotée par une spécification JML ;
- une version en ANSI-C annotée par la même spécification mais en ACSL.

Les deux versions ont les mêmes numéros de lignes d'instructions, notamment des erreurs. La précondition spécifie le contre-exemple employé pour le programme. Nous avons considéré qu'au plus trois conditions pouvaient être fausses sur un chemin. Par ailleurs, nous n'avons pas cherché de MCSs de cardinalité supérieure à 3. Nous ne cherchons donc pas tous les MCSs et notre approche n'est pas complète.

`BugAssist` utilise l'outil de **bounded model checking** CBMC [Clarke 2004] [5] pour générer la trace erronée et les données d'entrée. Cet outil est proche de CPBPV [Collavizza 2009], la différence principale est qu'il génère une formule conditionnelle purement booléenne et utilise un solveur SAT pour la résoudre. Pour le solveur MaxSAT, nous avons utilisé MSUnCore2 [Marques-Silva 2009]. Contrairement à `LocFaults`, `BugAssist` calcule tous les MCSs.

Toutes les expériences ont été effectuées avec un processeur Intel Core i7-3720QM 2.60 GHz avec 8 GO de RAM 64-bit Linux. `LocFaults` utilise les solveurs IBM CP OPTIMIZER and CPLEX [6].

5.4.3 Résultats sur des programmes sans boucles et sans calculs non linéaires

Nous avons premièrement testé notre implémentation de `LocFaults` et de l'outil `BugAssist` sur un ensembles de programmes sans boucles et sans calculs non linéaire. Les tables 5.1, 5.2, 5.3 contiennent les résultats que nous avons obtenus :

- Pour `LocFaults`, les chiffres sont les numéros de ligne, les numéros soulignés sont les conditions et les chiffres rouges sont les erreurs injectées qui ont été trouvées par les outils. Pour `LocFaults`, une ligne correspond à un chemin à travers le CFG, et contient soit une seule condition ou des conditions et les affectations avant la condition qui permettent de changer la branche. Par exemple, pour `TritypeV1` dans la colonne = 1 où une seule condition est déviée, `LocFaults` localise d'abord la condition 26 comme étant erronée. Puis il localise la condition 48 et localise les affectations 30 ou 25 qui peuvent être responsables de la mauvaise décision sur 48.
- Pour `BugAssist` les résultats correspondent à la fusion de l'ensemble des compléments des MSSs calculés, fusion qui est opérée par `BugAssist` avant l'affichage des résultats.

5. Voir http ://www.cprover.org/cbmc.
6. http://www-01.ibm.com/software/commerce/optimization/cplex-cp-optimizer/

Programme	Contre-exemple	Erreurs	LocFaults				BugAssist
			$= 0$	$= 1$	$= 2$	$= 3$	
AbsMinusKO	$\{i = 0, j = 1\}$	17	{17}	/	/	/	{17}
AbsMinusKO2	$\{i = 0, j = 1\}$	11	{11},{17}	/	/	/	{17, 20, 16}
AbsMinusKO3	$\{i = 0, j = 1\}$	14	{20}	{16},{14},{12}	/	/	{16, 20}
AbsMinusV2KO	$\{i = 0, j = 1\}$	13	{13}	/	/	/	{13}
AbsMinusV2KO2	$\{i = 0, j = 1\}$	11	{11},{13}	/	/	/	{13, 16, 20}
MinmaxKO	$\{in_1 = 2, in_2 = 1, in_3 = 3\}$	19	{10},{19}	{18},{10}	/	/	{14, 19, 30}
MidKO	$\{a = 2, b = 1, c = 3\}$	19	{19}	/	/	{14, 23, 26}	{14, 19, 30}
Maxmin6varKO	$\{a = 1, b = -4, c = -3, d = -1, e = 0, f = -4\}$	27	{28}	{15} {27}	/	/	{15, 12, 27 31, 166}
Maxmin6varKO2	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12	{65}	{12}	/	/	{12, 64, 166}
Maxmin6varKO3	$\{a = 1, b = -3, c = 0, d = -2, e = -1, f = -2\}$	12,15	{65}	/	{12, 15}	/	{12, 15, 64 166}
Maxmin6varKO4	$\{a = 1, b = -3, c = -4, d = -2, e = -1, f = -2\}$	12,15 19	{116}	/	/	{12, 15, 19}	{12, 166}
TritypeKO	$\{i = 2, j = 3, k = 2\}$	54	{54}	{26} {48},{30},{25}	{29, 32} {53, 57},{30},{25}	/	{26, 27, 32 33, 36, 48 57, 68}
TritypeKO2	$\{i = 2, j = 2, k = 4\}$	53	{54}	{21} {26} {35},{27},{25} {53},{27},{25}	{29, 57} {32, 44}	/	{21, 26, 27, 29, 30, 32, 33, 35, 36, 53, 68}

TABLE 5.1 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et sans calcul non linéaire. Ce tableau présente aussi le résultat de `BugAssist`.

TritypeKO2V2	$\{i=1,j=2,k=1\}$	31	{50}	{21} {26} {29} {36},{31},{25} {49},{31},{25}	{33, 45}	/	{21, 26, 27, 28, 31, 33, 34, 36, 37, 49, 68}
TritypeKO3	$\{i=1,j=2,k=1\}$	53	{54}	{21} {29} {35},{30},{25} {53},{30},{25}	{26, 57} {32, 44}	/	{21, 26, 27, 29, 30, 32, 33, 35, 36, 44, 53, 68}
TritypeKO4	$\{i=2,j=3,k=3\}$	45	{46}	{45},{33},{25}	{26, 32}	{32, 35, 49} {32, 35, 53} {32, 35, 57}	{26, 27, 29, 30, 32, 33, 35, 45, 49, 68}
TritypeKO5	$\{i=2,j=3,k=3\}$	32,45	{40}	{26} {29}	{32, 45} {35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TritypeKO6	$\{i=2,j=3,k=3\}$	32,33	{40}	{26} {29}	{35, 49},{25} {35, 53},{25} {35, 57},{25}	/	{26, 27, 29, 30, 32, 33, 35, 49, 68}
TriPerimetreKO	$\{i=2,j=1,k=2\}$	58	{58}	{31} {37},{32},{27}	/	/	{28, 29, 31, 32, 35, 37, 65, 72}
TriPerimetreKOV2	$\{i=2,j=3,k=2\}$	34	{34},{60}	{32} {37},{33},{27}	/	/	{28, 32, 33, 34, 36, 38, 40, 41, 52, 55, 56, 60, 64, 67, 74}

TABLE 5.2 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et sans calcul non linéaire. Ce tableau présente aussi le résultat de `BugAssist` (suite).

TriPerimetreKO2	$\{i=1, j=1, k=2\}$	57	{58}	{22} {28} {37},{29},{27} {57},{29},{27}	{31, 61} {34, 48}	/	{22, 28, 29, 31, 32, 34, 35, 37, 38, 48, 49, 52, 53, 57, 58, 61, 72}
TriPerimetreKO2V2	$\{i=1, j=2, k=1\}$	33	{54}	{22} {28} {31} {38},{33},{27} {53},{33},{27}	{35, 49}	/	{22, 28, 72 54, 53, 39, 33, 36, 38, 29, 31, 35, 49, 50}
TriPerimetreKO3	$\{i=2, j=1, k=2\}$	57	{58}	{22} {31} {37},{32},{27} {57},{32},{27}	/	/	{22, 28, 29, 31, 32, 34, 35, 37, 38, 49, 52, 57, , 72}
TriPerimetreKO4	$\{i=2, j=3, k=3\}$	49	{50}	{34} {37},{35},{27} {49},{35},{27}	/	/	{37, 35, 72, 50, 49, 34, 28, 29, 32, 61, 65, 31}
TriPerimetreKO5	$\{i=2, j=2, k=3\}$	34,49	{50}	{34} {37},{35},{27},{29}	{49, 54},{35},{27},{29}	/	{37, 35, 32, 29, 72, 34, 31, 49, 53}
TriPerimetreKO6	$\{i=2, j=2, k=3\}$	34,35	{50}	{34} {37},{35},{27},{29}	{49, 53},{35},{27},{29}	/	{37, 72, 29, 32, 35, 34, 31, 49, 53}

TABLE 5.3 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et sans calcul non linéaire. Ce tableau présente aussi le résultat de `BugAssist` (suite).

Programs	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
AbsMinusKO	0.706	0.021	0.022	0.025	0.026	0.02	0.03
AbsMinusKO2	0.692	0.029	0.042	0.042	0.035	0.02	0.06
AbsMinusKO3	0.693	0.021	0.042	0.037	0.038	0.02	0.03
AbsMinusV2KO	0.678	0.023	0.02	0.022	0.023	0.01	0.02
AbsMinusV2KO2	0.691	0.026	0.028	0.029	0.029	0.01	0.04
MinmaxKO	0.675	0.063	0.071	0.068	0.08	0.02	0.06
Maxmin6varKO	0.779	0.032	0.05	0.043	0.048	0.06	1.48
Maxmin6varKO2	0.781	0.028	0.046	0.04	0.042	0.07	0.91
Maxmin6varKO3	0.768	0.03	0.029	0.041	0.044	0.07	1.66
Maxmin6varKO4	0.785	0.029	0.03	0.034	0.05	0.07	1.05
TritypeKO	0.722	0.023	0.067	0.114	0.157	0.02	0.42
TritypeKO2	0.718	0.023	0.145	0.164	0.128	0.03	0.90
TritypeKO2V2	0.704	0.024	0.093	0.093	0.092	0.02	0.74
TritypeKO3	0.692	0.021	0.124	0.13	0.159	0.02	0.84
TritypeKO4	0.722	0.023	0.063	0.073	0.099	0.02	0.30
TritypeKO5	0.725	0.022	0.034	0.144	0.174	0.03	0.36
TritypeKO6	0.724	0.022	0.032	0.132	0.146	0.02	0.30
TriPerimetreKO	0.726	0.025	0.059	0.063	0.074	0.03	0.85
TriPerimetreKOV2	0.73	0.064	0.16	0.145	0.157	0.03	1.69
TriPerimetreKO2	0.751	0.025	0.115	0.155	0.127	0.04	2.32
TriPerimetreKO2V2	0.73	0.025	0.121	0.13	0.122	0.04	1.98
TriPerimetreKO3	0.728	0.027	0.122	0.146	0.143	0.03	1.67
TriPerimetreKO4	0.727	0.025	0.117	0.124	0.102	0.04	1.12
TriPerimetreKO5	0.727	0.024	0.106	0.159	0.181	0.04	1.00
TriPerimetreKO6	0.74	0.024	0.096	0.156	0.178	0.03	0.80

TABLE 5.4 – Temps de calcul pour les programmes sans boucles et sans contraintes non-linéaires.

Sur ces benchmarks les résultats de `LocFaults` sont plus concis et plus précis que ceux de `BugAssist`.

La table 5.4 fournit les temps de calcul : dans les deux cas, P correspond au temps de prétraitement et L au temps de calcul des MCSs. Pour `LocFaults`, le temps de pré-traitement inclut la traduction du programme Java en un arbre de syntaxe abstraite avec l'outil JDT (Eclipse Java development tools), ainsi que la construction du CFG dont les noeuds sont des ensembles de contraintes. C'est la traduction Java qui est la plus longue. Pour `BugAssist`, le temps de prétraitement est celui la construction de la formule SAT. Globalement, les performances de `LocFaults` et `BugAssist` sont similaires.

Ces résultats sont très encourageants. Ils montrent que `LocFaults` est plus précis que `BugAssist` lorsque les erreurs sont sur le chemin du contre exemple ou dans une

des conditions du chemin du contre-exemple. Ceci provient du fait que `BugAssist` et `LocFaults` ne calculent pas exactement la même chose :

`BugAssist` calcule les compléments des différents sous-ensembles obtenus par `MaxSAT`, c'est à dire, des sous-ensembles de clauses satisfiables de cardinalité maximale. Certaines "erreurs" du programme ne vont pas être identifiées par `BugAssist` car les contraintes correspondantes ne figurent pas dans le complément d'un sous ensemble de clauses satisfiables de cardinalité maximale.

`LocFaults` calcule des MCSs, c'est à dire, le complément d'un sous-ensemble de clauses maximal (MSS), c'est à dire, auquel on ne peut pas ajouter d'autre clause sans le rendre inconsistant, mais qui n'est pas nécessairement de cardinalité maximale.

`BugAssist` identifie des instructions suspectes dans l'ensemble du programme alors que `LocFaults` recherche les instructions suspectes sur un ensemble de chemins du programme.

5.4.4 Résultats sur des programmes sans boucles et avec calculs non linéaires

Nous avons aussi expérimenté et comparé `LocFaults` et l'outil `BugAssist` sur ensembles de programmes sans boucles et avec des opérations arithmétiques non linéaires. Les versions utilisées sont celles de `TriMultPerimetre` et `Heron`. Les résultats de ces expériences sont présentés sur les tables 5.5, 5.6, 5.7.

TriMultPerimetreKO	$\{i=2, j=1, k=2\}$	58	{58}	{31} {37},{27},{32}	/	/	{22, 37, 53, 49, 29, 35, 32, 31, 28, 65, 34, 62}
TriMultPerimetreKO2	$\{i=1, j=1, k=2\}$	57	{58}	{22} {28} {37},{27},{29} {57},{29},{27}	{31, 61} {34, 48}	/	{22, 37, 72, 58, 38, 52, 57, 49, 35, 32, 29, 28, 31, 65, 34}
TriMultPerimetreKO2V2	$\{i=1, j=2, k=1\}$	32	{53}	{21} {27} {30} {37},{32},{26} {52},{26},{32}	{34, 48}	/	{21, 27, 71, 49, 52, 38, 53, 32, 35, 37, 28, 30, 34, 48}
TriMultPerimetreKO3	$\{i=1, j=2, k=1\}$	56	{57}	{21} {30} {36},{26},{31} {56},{31},{26}	{27, 60} {34, 48}	/	{21, 71, 56, 51, 37, 57, 31, 28, 36, 34, 30, 27, 33, 47}
TriMultPerimetreKO4	$\{i=2, j=3, k=3\}$	56	{49}	{33} {36},{26},{34} {48},{34},{26}	/	/	{36, 34, 71, 49, 48, 33, 27, 28, 31, 53, 30, 60}
TriMultPerimetreKO5	$\{i=2, j=2, k=3\}$	33,48	{49}	{33} {36},{34},{28},{26}	{48, 52},{26},{34},{28}	/	{36, 34, 31, 28, 71, 49, 33, 30, 48 52}

TABLE 5.5 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et avec calculs non linéaires. Ce tableau présente aussi le résultat de `BugAssist`.

TriMultPerimetreKO6	$\{i=2, j=2, k=3\}$	33,34	{48}	{33} {36},{34},{28},{26}	{47, 51},{26},{34},{28}	/	{36, 70, 48, 28, 31, 34, 33, 30, 47 51}
HeronKO	$\{i=3, j=4, k=3\}$	61	{61}	{29} {35},{30},{25}	/	/	{19, 61, 79, 35, 27, 33, 30, 42, 29, 26, 71, 32, 48, 51, 54}
HeronKO2	$\{i=2, j=2, k=4\}$	59	{19},{62}	{26} {35},{27},{25} {59},{27},{25}	{29, 65} {32, 46}	/	{62, 80, 19, 59, 36, 42, 33, 35, 30, 27, 26, 29, 68, 32, 48, 51, 54}
HeronV1	$\{i=3, j=4, k=3\}$	61	{61}	{29} {35},{30},{25}	/	/	{79, 33, 30, 42, 35, 27, 61, 29, 26, 71, 32, 48, 36, 51}
HeronV2	$\{i=2, j=2, k=4\}$	59	{62}	{26} {35},{25},{27} {59},{27},{25}	{29, 65} {32, 46}	/	{62, 80, 59, 36, 42, 33, 35, 30, 27, 26, 29, 72, 32, 48, 51, 54}
HeronK02V2	$\{i=1, j=2, k=1\}$	31	{55}	{26} {29} {36},{25},{31} {52},{31},{25}	{33, 47}	/	26, 19, 52} 80, 55, 43, 31, 34, 36, 27, 29, 33, 47, 49}

TABLE 5.6 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et avec calculs non linéaires. Ce tableau présente aussi le résultat de `BugAssist` (suite).

HeronKO3	$\{i=1, j=2, k=1\}$	59	{62}	{29} {35},{25},{30} {59},{30},{25}	{26, 65} {32, 46}	/	80, 42, 19, } 59, 51, 62, 30, 27, 35, 33, 29, 26, 32, 46}
HeronKO4	$\{i=2, j=3, k=3\}$	47	{49}	{32} {35},{33},{25} {47},{33},{25}	/	/	{35, 33, 80, 49, 47, 32, 26, 19, 27, 30, 55, 29}
HeronKO5	$\{i=2, j=2, k=3\}$	32,47	{49}	{20} {32} {35},{33},{25},{27}	{47, 52},{25},{33},{27}	/	{20, 35, 33, 30, 27, 80, 49, 32, 29, 47, 52}
HeronKO6	$\{i=2, j=2, k=3\}$	32,33	{48}	{20} {32} {35},{33},{27},{25}	{46, 51},{27},{33},{25}	/	{20, 35, 79, 48, 27, 30, 33, 32, 29, 46, 51}

TABLE 5.7 – Les MCSs identifiés par `LocFaults` pour les programmes sans boucles et avec calculs non linéaires. Ce tableau présente aussi le résultat de `BugAssist` (suite).

Programs	LocFaults					BugAssist	
	P	L				P	L
		$= 0$	≤ 1	≤ 2	≤ 3		
TriMultPerimetreKO	0.749	0.056	0.133	0.14	0.148	0.05	3.50
TriMultPerimetreKO2	0.739	0.052	0.21	0.231	0.272	0.05	5.54
TriMultPerimetreKO2V2	0.733	0.059	0.267	0.27	0.264	0.06	4.44
TriMultPerimetreKO3	0.731	0.065	0.226	0.247	0.259	0.06	4.03
TriMultPerimetreKO4	0.767	0.055	0.154	0.164	0.164	0.06	2.97
TriMultPerimetreKO5	0.756	0.056	0.113	0.166	0.177	0.05	4.09
TriMultPerimetreKO6	0.764	0.049	0.114	0.179	0.173	0.05	2.89
HeronKO	0.797	0.119	0.188	0.186	0.207	0.06	7.93
HeronKO2	0.791	0.058	0.202	0.216	0.215	0.07	7.83
HeronV1	0.781	0.057	0.115	0.121	0.124	0.07	11.96
HeronV2	0.77	0.049	0.191	0.213	0.213	0.07	8.29
HeronKO2V2	0.792	0.11	0.212	0.226	0.265	0.08	6.17
HeronKO3	0.738	0.101	0.24	0.248	0.27	0.07	7.71
HeronKO4	0.771	0.049	0.163	0.183	0.169	0.07	4.63
HeronKO5	0.755	0.05	0.119	0.186	0.178	0.07	5.45
HeronKO6	0.746	0.048	0.115	0.176	0.179	0.07	5.03

TABLE 5.8 – Temps de calcul pour les programmes sans boucles et avec contraintes non-linéaires.

Les programmes `TriMultPerimetre` et `Heron` sont quelques variations du programme `Tritype` qui retournent des expressions non-linéaires. Ils ont la même structure de contrôle que le programme original. `TriMultPerimetre` calcule le produit des trois côtés et `Heron` calcule le carré de la surface du triangle. La spécification de `Heron` est la formule de Heron : $\sqrt{s(s-i)(s-j)(s-k)}$ où $s = (i+j+k)/2$. La valeur renvoyée varie en fonction du type de triangle. Par exemple, si le triangle est isocèle et $i == j$, la valeur retournée est $s(s-i)(s-i)(s-k)$ et si le triangle est équilatéral, la valeur retournée est $(3 \times i^4)/16$. Pour veiller à ce que la valeur retournée est un entier, nous calculons le carré de la surface et nous supposons comme précondition que l'expression ci-dessus (s) est un entier.

Les temps de calcul sont très courts pour tous les programmes, sauf pour les versions de `TriMultPerimetre` et `Heron`. La table 5.8 indique les temps pour ces deux benchmarks. P représente toujours le temps de pré-traitement, tandis que les autres lignes contiennent le temps de résolution. `LocFaults` est plus rapide que `BugAssist` sur ces deux benchmarks. Cela montre clairement l'avantage d'utiliser un solveur de contraintes pour les programmes contenant des instructions numériques non-triviales.

Sur ces benchmarks, la taille de l'ensemble d'instructions suspectes identifiées par `BugAssist` est similaire à la somme des tailles des ensembles d'instructions suspectes générées par `LocFaults`.

5.4.5 Résultats sur le benchmark TCAS

Prog	Nb_E	Nb_CE	LF			BA
			≤ 1	≤ 2	≤ 3	
V1	1	131	131	131	131	131
V2	2	67	67	67	67	67
V3	1	23	23	23	23	13
V4	1	20	4	4	4	20
V5	1	10	9	9	9	10
V6	1	12	11	11	11	12
V7	1	36	36	36	36	36
V8	1	1	1	1	1	1
V9	1	7	7	7	7	7
V10	2	14	12	12	12	14
V11	2	14	12	12	12	14
V12	1	70	45	45	45	48
V13	1	4	4	4	4	4
V14	1	50	50	50	50	50
V16	1	70	70	70	70	70
V17	1	35	35	35	35	35
V18	1	29	28	28	28	29
V19	1	19	18	18	18	19
V20	1	18	18	18	18	18
V21	1	16	16	16	16	16
V22	1	11	11	11	11	11
V23	1	41	41	41	41	41
V24	1	7	7	7	7	7
V25	1	3	2	2	2	3
V26	1	11	7	7	7	11
V27	1	10	9	9	9	10
V28	1	75	74	74	74	58
V29	1	18	17	17	17	14
V30	1	57	57	57	57	57
V34	1	77	77	77	77	77
V35	1	75	74	74	74	58
V36	1	122	120	120	120	122
V37	1	94	21	22	22	94
V39	1	3	2	2	2	3
V40	2	122	0	72	72	122
V41	1	20	16	16	16	20

TABLE 5.9 – Nombre d'erreurs localisées pour TCAS.

La table 5.9 donne les résultats pour les programmes de la suite `TCAS`. La colonne Nb_E indique pour chaque programme le nombre d'erreurs qui ont été introduites dans le programme alors que la colonne Nb_CE donne le nombre de contre-exemples. Les colonnes LF et BA indiquent respectivement le nombre de contre-exemples pour lesquels `LocFaults` et `BugAssist` ont identifié l'instruction erronée. Les résultats présentés pour `LocFaults` ont été obtenus avec une déviation au plus ; sauf pour la version `V41` où deux déviations sont nécessaires.

La taille de l'ensemble des instructions suspectes identifiées par `BugAssist` est en général plus grande que la somme des tailles des ensembles d'instructions suspectes générées par `LocFaults` mais `BugAssist` identifie un peu plus d'erreurs que `LocFaults`. Plus précisément, comme `LocFaults` affiche un ensemble de MCSs pour chaque chemin erroné, le processus de localisation d'erreur est beaucoup plus facile qu'avec l'ensemble unique d'erreurs suspectes signalées par `BugAssist`.

En tout, les temps de calcul de `BugAssist` et `LocFaults` sont très similaires et inférieurs à une seconde pour ce benchmark qui ne contient quasiment aucune instruction arithmétique, bien adapté pour un solveur booléenne.

5.4.6 Conclusion

Ces benchmarks montrent que `LocFaults` produit des informations pertinentes et utiles sur chaque chemin erroné détecté. Ils montrent aussi que le processus de débogage est beaucoup plus facile avec les ensembles de petite taille fournis par `LocFaults` qu'avec l'ensemble global des instructions suspectes calculé par `BugAssist`. Notre approche est basée sur les flux, elle génère les ensembles d'instructions suspectes de façon incrémentale. Elle trouve des erreurs sur le chemin du contre-exemple, et rapporte quelques explications dans un ordre qui peut aider l'utilisateur à trouver le bug. Ceci est plus approprié pour le débogage qu'une approche globale comme `BugAssist` qui fusionne toutes les instructions suspectes dans un seul ensemble.

Pour les versions de `TriMultPerimetre` et `Heron`, contenant des instructions numériques non linéaires, `LocFaults` fournit un résultat de qualité dans un délai raisonnable. Cela montrent le bénéfice de l'utilisation d'un solveur de contraintes pour des programmes avec calculs numériques.

Nos expérimentations sur les programmes TCAS montrent que `LocFaults` se compare très favorablement à `BugAssist` pour ce benchmark, bien adapté à SAT. Cela signifie que notre cadre sur la base de contraintes fournit un bon moyen, simple et efficace, pour mélanger les contraintes booléennes et numériques.

5.4.7 Résultats sur des programmes avec boucles

Ces benchmarks servent à mesurer l'extensibilité de `LocFaults` par rapport à `BugAssist` pour des programmes sans boucles, en fonction de l'augmentation du

nombre de dépliage b. Nous avons pris trois programmes avec boucles : BubbleSort, Sum et SquareRoot. Nous avons provoqué le bug *Off-by-one* dans chacun. Le benchmark, pour chaque programme, est créé en faisant augmenter le nombre de dépliage b. b est égal au nombre d'itérations effectuées par la boucle dans le pire des cas. Nous faisons aussi varier le nombre de conditions déviées pour `LocFaults` de 0 à 3. b' indique le nombre de dépliages nécessaires à `CPBPV` pour trouver l'erreur dans chaque programme. Le CFG construit par `LocFaults` est déplié b' fois au lieu de b fois, car l'outil `CPBPV` ne déplie pas le programme plus qu'il faut pour sortir de la boucle. Même si l'outil de `LocFaults` est exécuté en spécifiant, dans la ligne de commande par exemple, que le nombre de dépliage est égal à b.

Nous avons utilisé le solveur MIP de CPLEX pour BubbleSort. Pour Sum et SquareRoot, nous avons fait collaborer les deux solveurs de CPLEX (CP et MIP) lors du processus de la localisation. En effet, lors de la collecte des contraintes, nous utilisons une variable pour garder l'information sur le type du CSP construit. Quand `LocFaults` détecte un chemin erroné[7] et avant de procéder au calcul des MCSs, il prend le bon solveur selon le type du CSP qui correspond à ce chemin : s'il est non linéaire, il utilise le solveur CP OPTIMIZER ; sinon, il utilise le solveur MIP.

Pour chaque benchmark, nous avons présenté un extrait de la table contenant les temps de calcul (les colonnes P et L affichent respectivement les temps de prétraitement et de calcul des MCSs), ainsi que le graphe qui correspond au temps de calcul des MCSs.

7. Un chemin erroné est celui sur lequel nous identifions les MCSs.

5.4.7.1 Le benchmark BubbleSort

Programs	b	b'	LocFaults					BugAssist	
			P	L				P	L
				$= 0$	≤ 1	≤ 2	≤ 3		
BubbleSortV0	4	2	1.268	0.561	0.553	0.508	0.948	0.34	55.27
BubbleSortV1	5	3	0.781	0.597	0.627	0.762	1.331	0.22	125.40
BubbleSortV2	6	4	0.764	1.461	1.496	1.75	4.118	0.41	277.14
BubbleSortV3	7	5	0.774	0.813	0.852	1.468	12.67	0.53	612.79
BubbleSortV4	8	6	0.838	4.787	4.911	6.01	116.347	1.17	1074.67
BubbleSortV5	9	7	0.837	14.234	14.228	16.753	492.178	1.24	1665.62
BubbleSortV6	10	8	0.866	27.389	27.608	33.573	2078.445	1.53	2754.68
BubbleSortV7	11	9	0.876	56.008	62.198	69.591	4916.434	3.94	7662.90
BubbleSortV8	12	10	0.95	126.439	126.233	157.238	/	/	/
BubbleSortV9	13	11	0.917	235.282	244.805	282.796	/	/	/
BubbleSortV10	14	12	0.91	363.627	360.651	500.626	/	/	/
BubbleSortV11	15	13	0.969	437.994	438.549	715.594	/	/	/
BubbleSortV12	16	14	0.976	591.28	621.072	971.357	/	/	/
BubbleSortV13	17	15	1.019	737.541	739.541	1726.373	/	/	/
BubbleSortV14	18	16	1.038	954.475	1023.731	2197.53	/	/	/
BubbleSortV15	19	17	1.078	1230.099	1305.219	3477.862	/	/	/
BubbleSortV16	20	18	3.124	3647.636	4495.171	/	/	/	/
BubbleSortV17	21	19	2.458	4698.388	4316.524	/	/	/	/
BubbleSortV18	22	20	2.667	6580.013	6669.919	/	/	/	/

TABLE 5.10 – Le temps de calcul pour le benchmark BubbleSort

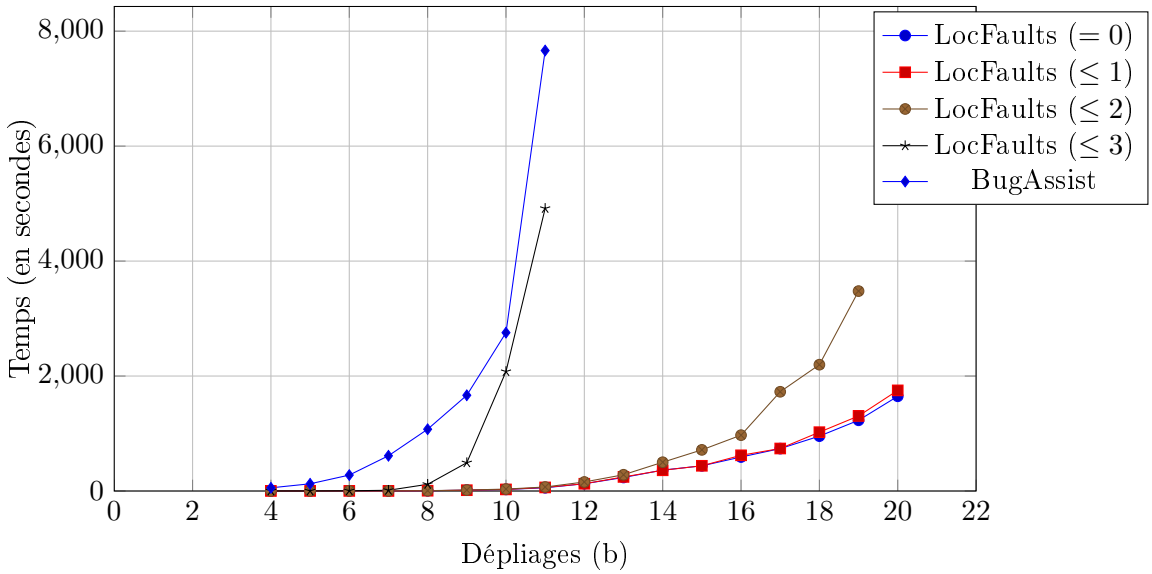


FIGURE 5.13 – Comparaison de l'évolution des temps des différentes versions de `LocFaults` et de `BugAssist` pour le benchmark BubbleSort, en faisant augmenter le nombre d'itérations en dépliant la boucle.

DCMs	*MCSs*
$\emptyset$	{5},{6},{9 : 1.11}, {9 : 2.11},{9 : 3.11}, {9 : 4.11},{9 : 5.11},{9 : 6.11},{9 : 7.11},{13}
{9 : 7}	{5},{6},{7},{9 : 1.10},{9 : 2.10},{9 : 3.10}, {9 : 4.10},{9 : 5.10}, {9 : 6.10},{9 : 1.11}, {9 : 2.11},{9 : 3.11},{9 : 4.11},{9 : 5.11}, {9 : 6.11}

TABLE 5.11 – MCD (Minimal Correction Deviation) et MCSs calculés par `LocFaults` pour le programme SquareRoot.

Les temps de `LocFaults` et `BugAssist` pour le benchmark BubbleSort sont présentés dans la table 5.10. Le graphe qui illustre l'augmentation des temps des différentes versions de `LocFaults` et de `BugAssist` en fonction du nombre de dépliages est donné dans la figure 5.13.

La durée d'exécution de `LocFaults` et de `BugAssist` pour ce benchmark croît exponentiellement avec le nombre de dépliages ; les temps de `BugAssist` sont toujours les plus grands. On peut considérer que `BugAssist` est inefficace pour ce benchmark. Les différentes versions de `LocFaults` (avec au plus 3, 2, 1 et 0 conditions déviées) restent utilisables jusqu'à un certain dépliage. Le nombre de dépliage au-delà duquel la croissance des temps de `BugAssist` devient rédhibitoire est inférieur à celui de `LocFaults`. Celui de `LocFaults` avec au plus 3 conditions déviées est inférieur à celui de `LocFaults` avec au plus 2 conditions déviées qui est inférieur lui aussi à celui de `LocFaults` avec au plus 1 conditions déviées. Les temps de `LocFaults` avec au plus 1 et 0 condition déviée sont presque les mêmes.

5.4.7.2 Les benchmarks SquareRoot et Sum

Avec un dépliage égal à 50, `BugAssist` calcule pour ce programme les instructions suspectes suivantes : $\{9, 10, 11, 13\}$. Le temps de la localisation est $36,16s$ et le temps de prétraitement est $0,12s$.

`LocFaults` présente une instruction suspecte en indiquant à la fois son emplacement dans le programme (la ligne d'instruction), ainsi que la ligne de la condition et l'itération de chaque boucle menant à cette instruction. Par exemple, 9 : 2.11 correspond à l'instruction qui se trouve à la ligne 11 dans le programme, cette dernière est dans une boucle dont la ligne de la condition d'arrêt est 9 et le numéro d'itération est 2. Les ensembles soupçonnés par `LocFaults` sont fournis dans la table 5.11.

Le temps de prétraitement est $0,769s$. Le temps écoulé lors de l'exploration du CFG et le calcul des MCS est $1,299s$. Nous avons étudié le temps de `LocFaults` et `BugAssist` des valeurs de *val* allant de 10 à 100 (le nombre de dépliage b employé en lançant les deux outils est égal à *val*), pour étudier le comportement combinatoire de chaque outil pour ce programme.

Programs	b	b'	LocFaults					BugAssist	
			P	L				P	L
				$= 0$	≤ 1	≤ 2	≤ 3		
V0	10	3	1.096	1.737	2.098	2.113	2.066	0.05	3.51
V10	20	4	0.724	0.974	1.131	1.117	1.099	0.05	6.54
V20	30	5	0.771	1.048	1.16	1.171	1.223	0.08	12.32
V30	40	6	0.765	1.048	1.248	1.266	1.28	0.09	23.35
V40	50	7	0.769	1.089	1.271	1.291	1.299	0.12	36.16
V50	60	8	0.741	1.041	1.251	1.265	1.281	0.14	38.22
V70	80	10	0.769	1.114	1.407	1.424	1.386	0.19	57.09
V80	90	11	0.744	1.085	1.454	1.393	1.505	0.22	64.94
V90	100	12	0.791	1.168	1.605	1.616	1.613	0.24	80.81

TABLE 5.12 – Le temps de calcul pour le benchmark SquareRoot

Programs	b	b'	LocFaults					BugAssist	
			P	L				P	L
				$= 0$	≤ 1	≤ 2	≤ 3		
V0	6	5	0.765	0.427	0.766	0.547	0.608	0.04	2.19
V10	16	15	0.9	0.785	1.731	1.845	1.615	0.08	17.88
V20	26	25	1.11	1.449	7.27	7.264	6.34	0.12	53.85
V30	36	35	1.255	0.389	8.727	4.89	4.103	0.13	108.31
V40	46	45	1.052	0.129	5.258	5.746	13.558	0.23	206.77
V50	56	55	1.06	0.163	7.328	6.891	6.781	0.22	341.41
V60	66	65	1.588	0.235	13.998	13.343	14.698	0.36	593.82
V70	76	75	0.82	0.141	10.066	9.453	10.531	0.24	455.76
V80	86	85	0.789	0.141	13.03	12.643	12.843	0.24	548.83
V90	96	95	0.803	0.157	34.994	28.939	18.141	0.31	785.64

TABLE 5.13 – Le temps de calcul pour le benchmark Sum

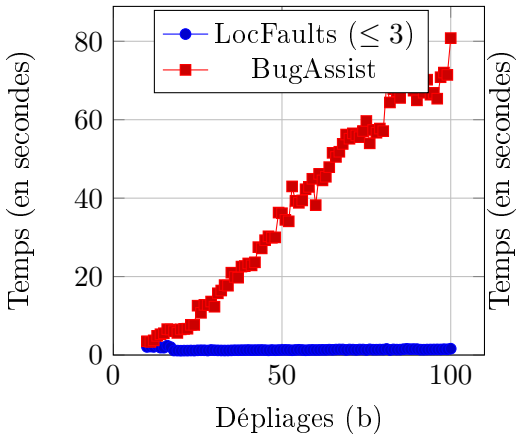


FIGURE 5.14 – Comparaison de l'évolution des temps de `LocFaultsV2` avec au plus 3 conditions déviées et de `BugAssist` pour le benchmark SquareRoot, en faisant augmenter le nombre d'itérations en dépliant la boucle.

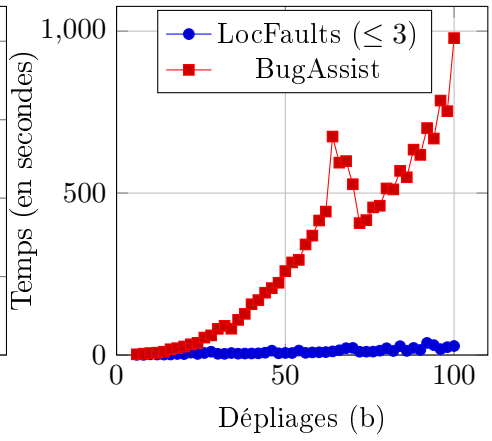


FIGURE 5.15 – Comparaison de l'évolution des temps de `LocFaultsV2` avec au plus 3 conditions déviées et de `BugAssist` pour le benchmark Sum, en faisant augmenter le nombre d'itérations en dépliant la boucle.

Les résultats en temps pour les benchmarks SquareRoot et Sum sont présentés dans les tableaux respectivement 5.12 et 5.13. Nous avons dessiné aussi le graphe qui correspond au résultat de chaque benchmark, voir respectivement le graphe de la figure 5.14 et 5.15. Le temps d'exécution de `BugAssist` croît rapidement ; les temps de `LocFaults` sont presque constants. Les temps de `LocFaults` avec au plus 0, 1 et 2 conditions déviées sont proches de ceux de `LocFaults` avec au plus 3 conditions déviées.

5.4.8 Conclusion

Nos résultats, pour les programmes avec boucles, montrent clairement la capacité de notre approche basée sur la programmation par contraintes et dirigée par les flots à maintenir ses performances face à l'augmentation du nombre de dépliage par rapport à `BugAssist`, si on limite le nombre de déviations. Naturellement, limiter le nombre de déviations peut empêcher de trouver certaines erreurs dans un programme. L'approche `BugAssist` calcule les MCSs sur la formule booléenne qui représente la totalité de la trace du contre-exemple. La taille du graphe global d'état de cette trace peut être (au moins) exponentielle en fonction de la taille du texte du programme [Emerson 2008]. Pour restreindre l'explosion combinatoire, notre approche profite de l'information du CFG déplié pour calculer les petits ensembles d'instructions suspectes, chacun d'eux étant associé à un chemin identifié. Sur le benchmark BubbleSort qui contient deux boucles imbriquées, pour de petites valeurs de b_{cond} (le nombre de déviations), le temps écoulé par `LocFaults` restera petit (même s'il est exponentiel en fonction du nombre de dépliage) et nous pourrons résoudre rapidement le problème. `BugAssist` est moins performant pour ce benchmark : les temps de `BugAssist` sont rédhibitoires et inutilisables.

5.5 Conclusion d'expérimentations

Ces expérimentations ont permis de valider notre approche sur un ensemble de benchmarks académiques. La comparaison avec l'outil `BugAssist` montre que pour des programmes avec calculs numériques, un solveur de contraintes est bien plus efficace qu'un solveur booléen. Les ensembles affichés par notre outil sont plus explicatifs et expressifs que ceux fournis par `BugAssist`, qui fusionne les instructions suspectes dans un ensemble unique ce qui rend la tâche de la correction d'erreurs compliquée pour l'utilisateur. Pour ces benchmarking, notre approche est plus efficace dans le sens où elle présente des résultats de qualité dans un délai raisonnable.

Quatrième partie

Conclusions et perspectives

Chapitre 6

Conclusions

"Rien ne fait peur à mon cœur plus d'un nombre à virgule flottante." – Gerald Jay Sussman

Sommaire

Dans ce chapitre, nous présentons la conclusion générale de notre travail, ainsi que les perspectives.

6.1 Conclusion

Afin d'analyser les erreurs dans un programme, un vérificateur de modèle permet d'avoir un historique d'exécution, appelé trace d'exécution. Les traces d'exécution produites comportent des informations concernant les chemins d'exécution. Souvent, les traces d'exécution comportent une quantité importante d'information, car les chemins d'exécution peuvent être très longs. Par conséquent, la localisation des portions de code qui contiennent des erreurs est un processus coûteux en temps et en effort, même pour des programmeurs expérimentés.

Dans ma thèse, nous avons proposé `LocFaults`, une nouvelle approche qui fournit une assistance à l'utilisateur pour localiser les erreurs à partir d'un contre-exemple. Notre approche combine les techniques de Bounded Model Checking avec un problème de satisfaction de contraintes. En effet, en partant d'un cas d'erreur (un contre-exemple), `LocFaults` explore le CFG du programme en profondeur et collecte les contraintes des instructions pour chaque sous-ensemble de chemins : le chemin initial et les chemins obtenus par les déviations. Quand un chemin erroné est

detecté, `LocFaults` isole les ensembles MCSs (Minimal Correction Subsets) d'une taille bornée inclus dans le CSP qui correspond au chemin pour identifier la suspicion dans ce dernier. Plus précisément, notre algorithme propage le contre-exemple sur le CFG pour explorer premièrement le chemin du contre-exemple. Ensuite, il entame la recherche des instructions suspectes sur d'autres chemins erronés en déviant au plus b_{cond} branchements par rapport au comportement du contre-exemple. Si le chemin résultant satisfait la postcondition (autrement dit, si la sortie du programme après la déviation du contre-exemple ne contredit plus la postcondition) il calcule les MCSs sur le chemin qui précède la dernière condition déviée ; sinon, le chemin est rejeté. Pour calculer les MCSs, notre outil utilise différents solveurs de contraintes. Il peut même faire collaborer plusieurs solveurs de contraintes lors du processus de localisation, afin d'utiliser le solveur le mieux performé en fonction du type du système de contraintes du chemin détecté pour accélérer la résolution. Nous avons implémenté et adapté pour cela un algorithme générique proposé par Liffiton et Sakallah afin de pouvoir traiter plus efficacement des programmes avec des calculs numériques.

Nous avons mené de nombreuses expérimentations, d'une part, pour tester par des expériences répétées la validité de notre approche, et d'autre part, pour démontrer sa compétitivité par rapport à l'approche `BugAssist`. Nous avons évalué notre approche sur un ensemble de programmes académiques et réalistes. Nous avons aussi comparé les performances de notre implémentation avec `BugAssist` sur ces ensembles de programmes. Les résultats obtenus ont montré la précision de notre outil, ainsi que son efficacité par rapport à `BugAssist` sur des programmes avec calculs numériques.

Dans les premiers résultats (voir la section 5.4.3), nous avons comparé notre outil et `BugAssist` sur quelques programmes sans boucles et sans calculs non linéaires. Ils ont montré que `LocFaults` localise avec plus de précision que `BugAssist` les erreurs lorsqu'elles sont sur le chemin du contre-exemple ou dans une des conditions du chemin du contre-exemple. Ces résultats montrent également que la localisation d'erreurs est plus aisée avec `LocFaults` qui fournit les instructions suspectes dans des ensembles séparés. `BugAssist` fournit en sortie un seul ensemble qui rassemble toutes les MCSs calculés. De plus, `LocFaults` fournit les résultats dans un ordre qui peut être utile à l'utilisateur pour trouver les erreurs. Les temps de calcul sont similaires pour les deux outils.

Les résultats sur les différentes variations du programme `Tritype` contenant des opérations arithmétiques non linéaires (voir la section 5.4.4) montrent que sur des programmes avec calculs numériques, notre approche produit un résultat à la fois de qualité et dans un laps de temps raisonnable. `BugAsisst` (qui utilise un solveur MaxSAT pour trouver les MCSs) est moins performant pour ce type de programmes. Ce qui signifie l'apport des contraintes pour traiter des programmes contenant des instructions numériques.

Les expérimentations sur le benchmark connu TCAS (voir la section 5.4.5) qui

ne contient que peu d'instructions numériques confirment que `LocFaults` est aussi un bon moyen pour localiser les erreurs dans des programmes contenant un mélange d'instructions booléennes et numériques, car les performances des deux outils sont très proches sur ce benchmark normalement plus approprié à un solveur SAT.

Les résultats de `LocFaults` par rapport à `BugAssist` sur un ensemble de programmes avec boucles (voir la section 5.4.7) ont montré que les temps de notre approche peuvent rester utilisables jusqu'à un certain dépliage. Le paramètre du nombre de déviations joue un rôle important à l'augmentation des temps de `LocFaults` pour le programme avec deux boucles imbriquées `BubbleSort` : le nombre de dépliage auquel la croissance des temps de calcul devient rédhibitoire augmente avec le nombre de déviations utilisé. Les temps de `BugAssist` sont toujours les plus grands que `LocFaults` pour une déviation inférieure à 3. Pour les programmes Sum et SquareRoot les temps sont presque constants et courts ; alors que pour `BugAssist`, ils croissent rapidement.

Pour terminer, `BugAssist` procède en traduisant tout le programme et l'assertion avec le contre-exemple en une formule booléenne. Pour trouver les instructions potentiellement erronées, il calcule les MCSs dans cette formule. L'outil `BugAssist` que nous avons expérimenté fournit les MCSs calculés dans un seul ensemble, ceci complique la tâche de la localisation d'erreurs pour l'utilisateur. Notre outil affiche les MCSs sur le chemin du contre-exemple et les chemins de déviations qui corrigent le programme, ce qui peut être très utile à l'utilisateur pour comprendre et corriger les bugs dans son programme. De plus, le calcul des MCSs dans la formule construite est résolu par un solveur MaxSAT, alors que notre méthode utilise les informations sur le chemin courant pour savoir s'il corrige le programme et donc calculer les MCSs sur le système de contraintes construit par l'usage d'un solveur de contraintes.

6.2 Perspectives

Dans le cadre de nos travaux futurs, nous envisageons de confirmer nos résultats sur des programmes avec plusieurs boucles complexes (voir nos premiers résultats dans [Bekkouche 2015a]). Nous envisageons de comparer les performances de `LocFaults` avec des méthodes statistiques existantes ; par exemple : Tarantula [Jones 2005, Jones 2002], Ochiai [Abreu 2007], AMPLE [Abreu 2007], Pinpoint [Chen 2002]. Usuellement, l'intuition et les connaissances d'un programmeur expérimenté concernant le programme et sur l'emplacement du bug devraient être explorées, pour le choix d'une stratégie de débogage efficace [Wong 2010]. Pour cela, nous développons une version interactive de notre outil qui fournit les sous-ensembles suspects l'un après l'autre : nous voulons tirer profit des connaissances de l'utilisateur pour sélectionner les conditions qui doivent être déviées. Il sera certainement aussi intéressant de calculer les IISs/MUSs car ils apportent une information complémentaire pour l'utilisateur.

6.2.1 Calcul des MUSs

Les raisons d'un CSP étant sur-contraint sont un ou plusieurs sous-ensembles de contraintes qui ne peuvent pas être satisfaites ensemble. Ces ensembles conflictuels de contraintes sont les MUSs/IISs. Une forme de diagnostic d'erreurs est de trouver tous les sous-ensembles insatisfiables minimes (MUSs), afin de générer une meilleure explication des erreurs pour l'utilisateur. En plus, un MUS représente un noyau infaisable, donc forcément qu'il doit couvrir une partie des erreurs. Pour restaurer la satisfaisabilité dans le CSP, il faut réparer chacun des MUSs. Pour aider à mieux expliquer et comprendre les conflits, nous envisageons donc de calculer les MUSs sur les chemins erronés détectés en utilisant les MCSs trouvés par `LocFaults`. Dans notre outil, nous avons implémenté les algorithmes `Deletion Filter`, `Additive Method` et la méthode qui les combine, ainsi que `QuickXplain`. Mais, un système de contraintes peut avoir plusieurs IISs/MUSs. Nous pourrons pour cela exporter et réutiliser les différents algorithmes calculant plusieurs IISs/MUSs cités dans notre état de l'art (voir la section 3.6), comme l'algorithme DAA [Bailey 2005] ou MARCO [Liffiton 2013].

6.2.2 Localisation d'erreurs pour les programmes avec calcul sur les flottants

Dans une société où la part des applications embarquées ne cesse de grandir et leur caractère critique d'augmenter, la capacité à en vérifier la correction et la robustesse devient un enjeu majeur. C'est un enjeu particulièrement crucial pour le calcul basé sur l'arithmétique des nombres à virgule flottante. En effet, cette arithmétique qui a des comportements assez inhabituels [Goldberg 1991, Monniaux 2008] est de plus en plus utilisée dans les logiciels embarqués. Par exemple, pour certaines valeurs d'entrée, le flot de contrôle lié aux réels peut passer dans une branche de la conditionnelle tandis qu'il passe par l'autre pour les flottants [Goubault 2013]. Ces contributions ont montré l'intérêt des approches par contraintes pour vérifier la conformité d'un programme vis-à-vis de sa spécification, mais aussi les limites de ces techniques. Notre objectif ici est l'aide à la localisation d'erreurs pour les programmes avec calculs sur les flottants. Le cadre fondé sur les contraintes que nous avons mis en place est bien adapté pour le traitement des opérations arithmétiques. Il peut être étendu pour la localisation d'erreurs dans les programmes contenant des instructions sur les flottants. Une telle extension serait plus difficile pour des approches qui se basent sur SAT comme `BugAssist`, ou `SNIPER` où l'ensemble du programme est transformée en une représentation intermédiaire LLVM.

Le défi est que la plupart des programmeurs ne sont pas des experts en virgule flottante [Demmel 2014]. Comme exemples d'anomalies communes : les erreurs d'arrondi qui peuvent accumuler trop dans un programme numérique, branchements conditionnels impliquant des comparaisons de nombres à virgule flottante qui peuvent aller égarés en raison des subtilités de l'arithmétique flottante, le débordement,

l'annulation catastrophique, etc [Bailey 2010].

Exemple [Ballard 2010] Prenons l'exemple du programme C suivant.

```
#include <stdio.h>
int main (){
 double a = 1e17;
 double b = 1;
 double c = a + b;
 double d = c - a;
 printf(" => total error: %1.16e\n", fabs(1.0-d));
 return 0;
}
```

En double précision, ce programme va calculer $1e17 + 1$ comme $1e17$ et puis donner à d la valeur 0. En arithmétique exacte (ou de précision double-double), d est égal à b, ou 1. L'erreur numérique se produit à l'affectation $c = a + b$, et deux changements sont nécessaires pour résoudre ce problème. Premièrement, l'ajout doit être fait dans une plus grande précision, et le second, c doit être stocké comme un double-double pour empêcher la perte d'informations.

6.2.3 Mesurer le degré de suspicion de chaque instruction

L'idée est d'exploiter les ensembles que nous fournissons pour mesurer la suspicion de chaque instruction dans chaque chemin erroné détecté, afin de mieux aider l'utilisateur à trouver les bugs dans son programme. Le problème revient à chercher une formule pour mesurer l'infaisabilité de chaque contrainte dans un ensemble de contraintes infaisable.

Pour comprendre la nature de l'incohérence et la quantifier dans un système de contraintes inconsistant, un certain nombre de fonctions de mesure d'incohérence ont été proposées. On trouve par exemple la fonction de mesure de **Shapley**[1] [Shapley 1952, Hunter 2006] dans le papier [Hunter 2008]. Soit C un ensemble de contraintes infaisable, et $\cup_{MUS}(C)$ l'ensemble de tous les MUSes inclus dans C. Nous cherchons à mesurer l'infaisabilité de chaque contrainte $c \in C$. Des formes simples d'une fonction mesurant l'infaisabilité (notée MIV(MinInc Inconsistency Value)) sont définies comme suit [Hunter 2008] :

- $MIV_D(C,c) = \begin{cases} 1 \text{ si } \exists\, mus \in \cup_{MUS}(C) \mid c \in mus \\ 0 \text{ dans le cas contraire} \end{cases}$
- $MIV_{card}(C,c) = |\{mus \in \cup_{MUS}(C) \mid c \in mus\}|$

La fonction MIV_D mesure l'infaisabilité en prenant comme critère l'appartenance de la contrainte à un MUS, en distinguant seulement les contraintes qui appartiennent (la valeur de la fonction est 1) et les contraintes qui n'appartiennent pas à cet ensemble (la valeur de la fonction est 0) ; en d'autre termes, il n'y a pas de distinction entre les contraintes qui appartiennent à au moins un MUS : elles

1. http://fr.wikipedia.org/wiki/Lloyd_Shapley

ont toutes comme degré d'infaisabilité 1. La fonction MIV_{card} calcule le nombre de MUSs qui contient la contrainte. La cardinalité de l'ensemble de contraintes dans un MUS est importante pour quantifier l'infaisabilité des contraintes. Par exemple, une contrainte qui appartient à un MUS de cardinalité k pourrait avoir un degré d'infaisabilité plus grand qu'une contrainte appartenant à un MUS de cardinalité inférieur à k. Le critère entre les cardinalités des MUSs n'est pas pris en compte par la fonction MIV_{card}. Voici une fonction utilisant ce critère [Hunter 2008] :

$$\_\ MIV_C(C,c) = \sum_{mus \in \cup_{MUS}(C) \mid c \in mus} \frac{1}{|mus|}$$

6.2.4 Méthode hybride, statistique et BMC, pour la localisation d'erreurs

Il y a beaucoup d'approches en test, seulement peu sont proposées dans le cadre du Model Checking et encore moins (voire aucune) combinant les deux pour traiter le problème de localisation d'erreurs. Pour but d'améliorer encore l'efficacité et la qualité des résultats affichés à l'utilisateur, nous prévoyons de développer une nouvelle approche qui combine le test et les techniques de BMC pour l'aide à la localisation d'erreurs. L'objectif est d'utiliser la puissance de notre approche pour calculer les sous-ensembles de correction minimales (MCSs) en analysant les chemins du CFG du programme, mais aussi la simplicité des approches a base de tests (comme l'approche statistique `Tarantula` [Jones 2001, Jones 2002, Jones 2005]) pour mesurer le degré de suspicion des instructions lors de l'exécution de plusieurs cas de tests.

6.2.5 Apprentissage automatique pour localiser les erreurs

L'idée de l'apprentissage automatique (par exemple, les arbres de décision, les machines à vecteurs de support (en anglais Support Vector Machine, SVM)) est de permettre de classifier ou de prédire le type des données. Par exemple, en imagerie, nous utilisons les approches de l'apprentissage automatique pour la reconnaissance et la classification de visages. Dans notre cadre, nous pouvons utiliser l'apprentissage automatique pour prédire si les instructions d'un programme donné sont suspectes ou non. Par exemple, nous préparons les données (les cas de tests erronés et réussis) et sélectionnons les caractéristiques (comme les informations obtenues à l'aide du CFG en utilisant l'outil CPBPV [Collavizza 2010] sur la couverture des instructions) pour construire un classificateur permettant de prédire le type de données (instruction suspecte ou non). Plusieurs techniques fondées sur l'apprentissage automatiques pour localiser les erreurs dans un programme ont été proposées. Citons, à titre d'exemple, Wong et al. [Wong 2009b] qui a proposé une méthode de localisation d'erreurs basée sur un algorithme de rétropropagation (BP pour Back Propagation) de réseau neuronal.

Table des figures

Liste des tableaux

List d'algorithmes

Bibliographie

[Abreu 2007] Rui Abreu, Peter Zoeteweij et Arjan JC Van Gemund. On the accuracy of spectrum-based fault localization. In Testing : Academic and Industrial Conference Practice and Research Techniques-MUTATION, 2007. TAICPART-MUTATION 2007, pages 89–98. IEEE, 2007. (Cité en pages 15 et 111.)

[Bailey 2005] James Bailey et Peter J Stuckey. Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In Practical Aspects of Declarative Languages, pages 174–186. Springer, 2005. (Cité en pages 24, 39, 40, 43, 46 et 112.)

[Bailey 2010] David H Bailey, James Demmel, William Kahan, Guillaume Revy et Koushik Sen. Techniques for the automatic debugging of scientific floating-point programs. 2010. (Cité en page 113.)

[Ball 2003] Thomas Ball, Mayur Naik et Sriram K. Rajamani. From symptom to cause : localizing errors in counterexample traces. In Proceedings of POPL, pages 97–105. ACM, 2003. (Cité en page 17.)

[Ballard 2010] Grey Ballard. Developing Tools for Floating-point Debugging. 2010. (Cité en page 113.)

[Barnett 2005] Mike Barnett et K Rustan M Leino. Weakest-precondition of unstructured programs. In ACM SIGSOFT Software Engineering Notes, volume 31, pages 82–87. ACM, 2005. (Cité en pages 52 et 55.)

[Beer 2009] Ilan Beer, Shoham Ben-David, Hana Chockler, Avigail Orni et Richard J. Trefler. Explaining Counterexamples Using Causality. In Proceedings of CAV, volume 5643 of Lecture Notes in Computer Science, pages 94–108. Springer, 2009. (Cité en page 18.)

[Bekkouche 2014] Mohammed Bekkouche, Hélène Collavizza et Michel Rueher. Une approche CSP pour l'aide\a la localisation d'erreurs. arXiv preprint arXiv :1404.6567, 2014. (Cité en pages 3 et 62.)

[Bekkouche 2015a] Mohammed Bekkouche. Exploration of the scalability of LocFaults approach for error localization with While-loops programs. arXiv preprint arXiv :1503.05508, 2015. (Cité en page 111.)

[Bekkouche 2015b] Mohammed Bekkouche, Hélène Collavizza et Michel Rueher. LocFaults : A new flow-driven and constraint-based error localization approach*. In SAC'15, SVT track, 2015. (Cité en pages 3 et 62.)

[Birnbaum 2003] Elazar Birnbaum et Eliezer L Lozinskii. Consistent subsets of inconsistent systems : structure and behaviour. Journal of Experimental & Theoretical Artificial Intelligence, vol. 15, no. 1, pages 25–46, 2003. (Cité en pages 23 et 27.)

[Chen 2002] Mike Y Chen, Emre Kiciman, Eugene Fratkin, Armando Fox et Eric Brewer. Pinpoint : Problem determination in large, dynamic internet services. In Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on, pages 595–604. IEEE, 2002. (Cité en pages 12 et 111.)

[Chinneck 1991] John W Chinneck et Erik W Dravnieks. Locating minimal infeasible constraint sets in linear programs. ORSA Journal on Computing, vol. 3, no. 2, pages 157–168, 1991. (Cité en page 29.)

[Chinneck 1996] John W. Chinneck. Localizing and Diagnosing Infeasibilities in Networks. INFORMS Journal on Computing, vol. 8, no. 1, pages 55–62, 1996. (Cité en page 22.)

[Chinneck 2001] John W. Chinneck. Fast Heuristics for the Maximum Feasible Subsystem Problem. INFORMS Journal on Computing, vol. 13, no. 3, pages 210–223, 2001. (Cité en pages 22 et 29.)

[Chinneck 2008] John W. Chinneck. Feasibility and infeasibility in optimization : Algorithms and computational methods. Springer, 2008. (Cité en pages 22, 29, 31 et 38.)

[Clarke 1982] Edmund M Clarke et E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. Springer, 1982. (Cité en page 17.)

[Clarke 2004] Edmund Clarke, Daniel Kroening et Flavio Lerda. A tool for checking ANSI-C programs. In Tools and Algorithms for the Construction and Analysis of Systems, pages 168–176. Springer, 2004. (Cité en page 89.)

[Collavizza 2009] Hélene Collavizza. Contribution à la vérication formelle et programmation par contraintes. PhD thesis, Université Nice Sophia Antipolis, 2009. (Cité en page 89.)

[Collavizza 2010] Hélène Collavizza, Michel Rueher et Pascal Van Hentenryck. CPBPV : a constraint-programming framework for bounded program verification. Constraints, vol. 15, no. 2, pages 238–264, 2010. (Cité en pages 55, 70, 89 et 114.)

[Collavizza 2014] Hélène Collavizza, Nguyen Le Vinh, Olivier Ponsini, Michel Rueher et Antoine Rollet. Constraint-based BMC : a backjumping strategy. STTT, vol. 16, no. 1, pages 103–121, 2014. (Cité en page 55.)

[De Kleer 1987] Johan De Kleer et Brian C Williams. Diagnosing multiple faults. Artificial intelligence, vol. 32, no. 1, pages 97–130, 1987. (Cité en page 41.)

[de la Banda 2003] Maria Garcia de la Banda, Peter J Stuckey et Jeremy Wazny. Finding all minimal unsatisfiable subsets. In Proceedings of the 5th ACM SIGPLAN international conference on Principles and practice of declaritive programming, pages 32–43. ACM, 2003. (Cité en pages 43 et 46.)

[Demmel 2014] Koushik Sen PI James Demmel et Costin Iancu. CORVETTE : Program Correctness, Verification, and Testing for Exascale. 2014. (Cité en page 112.)

[D'silva 2008] Vijay D'silva, Daniel Kroening et Georg Weissenbacher. A survey of automated techniques for formal software verification. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, vol. 27, no. 7, pages 1165–1178, 2008. (Cité en pages 17 et 62.)

[Emerson 2008] E Allen Emerson. The beginning of model checking : a personal perspective. In 25 Years of Model Checking, pages 27–45. Springer, 2008. (Cité en page 105.)

[Felfernig 2012] Alexander Felfernig, Monika Schubert et Christoph Zehentner. An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM, vol. 26, no. 1, pages 53–62, 2012. (Cité en pages 25, 26 et 28.)

[Goldberg 1991] David Goldberg. What every computer scientist should know about floating-point arithmetic. ACM Computing Surveys (CSUR), vol. 23, no. 1, pages 5–48, 1991. (Cité en page 112.)

[Gotlieb 1998] Arnaud Gotlieb, Bernard Botella et Michel Rueher. Automatic test data generation using constraint solving techniques. In ACM SIGSOFT Software Engineering Notes, volume 23, pages 53–62. ACM, 1998. (Cité en page 9.)

[Goubault 2013] Eric Goubault et Sylvie Putot. Robustness analysis of finite precision implementations. In Programming Languages and Systems, pages 50–57. Springer, 2013. (Cité en page 112.)

[Grégoire 2014] Eric Grégoire, Jean-Marie Lagniez et Bertrand Mazure. An experimentally efficient method for (MSS, CoMSS) partitioning. In Twenty-Eighth AAAI Conference on Artificial Intelligence, 2014. (Non cité.)

[Griesmayer 2006] Andreas Griesmayer, Roderick Bloem et Byron Cook. Repair of Boolean Programs with an Application to C. In Proceedings of CAV, volume 4144 of Lecture Notes in Computer Science, pages 358–371. Springer, 2006. (Cité en page 18.)

[Griesmayer 2007] Andreas Griesmayer, Stefan Staber et Roderick Bloem. Automated Fault Localization for C Programs. Electr. Notes Theor. Comput. Sci., vol. 174, no. 4, pages 95–111, 2007. (Cité en page 18.)

[Groce 2004] Alex Groce, Daniel Kroening et Flavio Lerda. Understanding Counterexamples with explain. In Proceedings of CAV, volume 3114 of Lecture Notes in Computer Science, pages 453–456. Springer, 2004. (Cité en page 17.)

[Groce 2005] Alex David Groce. Error explanation and fault localization with distance metrics. PhD thesis, NASA Ames Research Center, 2005. (Cité en page 17.)

[Groce 2006] Alex Groce, Sagar Chaki, Daniel Kroening et Ofer Strichman. Error explanation with distance metrics. STTT, vol. 8, no. 3, pages 229–247, 2006. (Cité en page 17.)

[Guieu 1999] Olivier Guieu et John W Chinneck. Analyzing infeasible mixed-integer and integer linear programs. INFORMS Journal on Computing, vol. 11, no. 1, pages 63–77, 1999. (Cité en pages 29 et 31.)

[Guo 2006] Liang Guo, Abhik Roychoudhury et Tao Wang. Accurately choosing execution runs for software fault localization. In Compiler Construction, pages 80–95. Springer, 2006. (Cité en page 15.)

[Han 1999] Benjamin Han et Shie-Jue Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. Systems, Man, and Cybernetics, Part B : Cybernetics, IEEE Transactions on, vol. 29, no. 2, pages 281–286, 1999. (Cité en page 39.)

[Hou 1994] Aimin Hou. A theory of measurement in diagnosis from first principles. Artificial Intelligence, vol. 65, no. 2, pages 281–328, 1994. (Cité en pages 39 et 43.)

[Hunter 2006] Anthony Hunter, Sébastien Konieczny et al. Shapley Inconsistency Values. KR, vol. 6, pages 249–259, 2006. (Cité en page 113.)

[Hunter 2008] Anthony Hunter, Sébastien Konieczny et al. Measuring Inconsistency through Minimal Inconsistent Sets. KR, vol. 8, pages 358–366, 2008. (Cité en pages 113 et 114.)

[Jones 2001] James A Jones, Mary Jean Harrold et John T Stasko. Visualization for fault localization. In Proceedings of ICSE 2001 Workshop on Software Visualization, Toronto, Ontario, Canada, pages 71–75. Citeseer, 2001. (Cité en pages 11 et 114.)

[Jones 2002] James A Jones, Mary Jean Harrold et John Stasko. Visualization of test information to assist fault localization. In Proceedings of the 24th international conference on Software engineering, pages 467–477. ACM, 2002. (Cité en pages 11, 74, 111 et 114.)

[Jones 2005] James A Jones et Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering, pages 273–282. ACM, 2005. (Cité en pages 10, 12, 111, 114 et 117.)

[Jose 2011a] Manu Jose et Rupak Majumdar. Bug-Assist : assisting fault localization in ANSI-C programs. In Computer Aided Verification, pages 504–509. Springer, 2011. (Cité en page 3.)

[Jose 2011b] Manu Jose et Rupak Majumdar. Bug-Assist : Assisting Fault Localization in ANSI-C Programs. In Proceedings of CAV, volume 6806 of Lecture Notes in Computer Science, pages 504–509. Springer, 2011. (Cité en pages 19, 70 et 72.)

[Jose 2011c] Manu Jose et Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. ACM SIGPLAN Notices, vol. 46, no. 6, pages 437–446, 2011. (Cité en page 3.)

[Jose 2011d] Manu Jose et Rupak Majumdar. Cause clue clauses : error localization using maximum satisfiability. In Proceedings of PLDI, pages 437–446. ACM, 2011. (Cité en pages 19, 70 et 72.)

[Junker 2004] Ulrich Junker. QUICKXPLAIN : Preferred Explanations and Relaxations for Over-Constrained Problems. In Proceedings of AAAI, pages 167–172. AAAI Press / The MIT Press, 2004. (Cité en pages 29, 31, 33 et 36.)

[Kilby 2005] Philip Kilby, John Slaney, Sylvie Thiébaux et Toby Walsh. Backbones and backdoors in satisfiability. In AAAI, volume 5, pages 1368–1373, 2005. (Cité en page 26.)

[Lamraoui 2014] Si-Mohamed Lamraoui et Shin Nakajima. A Formula-based Approach for Automatic Fault Localization in Imperative Programs. NII research report, Submitted For publication, 6 pages, February, 2014. (Cité en page 20.)

[LeBlanc 1987] Thomas J LeBlanc et John M Mellor-Crummey. Debugging parallel programs with instant replay. Computers, IEEE Transactions on, vol. 100, no. 4, pages 471–482, 1987. (Non cité.)

[Lewis 1973] David Lewis. Causation. The journal of philosophy, pages 556–567, 1973. (Cité en page 17.)

[Liblit 2005] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken et Michael I Jordan. Scalable statistical bug isolation. In ACM SIGPLAN Notices, volume 40, pages 15–26. ACM, 2005. (Cité en page 14.)

[Liffiton 2005] Mark H Liffiton et Karem A Sakallah. On finding all minimally unsatisfiable subformulas. In Theory and Applications of Satisfiability Testing, pages 173–186. Springer, 2005. (Cité en page 41.)

[Liffiton 2008] Mark H. Liffiton et Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. J. Autom. Reasoning, vol. 40, no. 1, pages 1–33, 2008. (Cité en pages 22, 23, 37, 38, 41, 43, 46, 47, 58, 65 et 70.)

[Liffiton 2009] Mark H Liffiton et Karem A Sakallah. Generalizing core-guided Max-SAT. In Theory and Applications of Satisfiability Testing-SAT 2009, pages 481–494. Springer, 2009. (Cité en page 41.)

[Liffiton 2013] Mark H. Liffiton et Ammar Malik. Enumerating Infeasibility : Finding Multiple MUSes Quickly. In Proc. of CPAIOR, volume 7874 of Lecture Notes in Computer Science, pages 160–175. Springer, 2013. (Cité en pages 43, 46 et 112.)

[Liffiton 2015] Mark H Liffiton, Alessandro Previti, Ammar Malik et Joao Marques-Silva. Fast, flexible MUS enumeration. Constraints, pages 1–28, 2015. (Cité en pages 40, 44 et 45.)

[Liu 2006] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han et Samuel P Midkiff. Statistical debugging : A hypothesis testing-based approach. Software Engi-

neering, IEEE Transactions on, vol. 32, no. 10, pages 831–848, 2006. (Cité en page 15.)

[Liu 2010] Yongmei Liu et Bing Li. Automated Program Debugging Via Multiple Predicate Switching. In Proceedings of AAAI. AAAI Press, 2010. (Cité en page 18.)

[Marques-Sila 2011] Joao Marques-Sila et Jordi Planes. Algorithms for maximum satisfiability using unsatisfiable cores. In Advanced Techniques in Logic Synthesis, Optimizations and Applications, pages 171–182. Springer, 2011. (Cité en page 19.)

[Marques-Silva 2009] Joao Marques-Silva. The msuncore maxsat solver. SAT, page 151, 2009. (Cité en pages 19 et 89.)

[Marques-Silva 2013] Joao Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti et Anton Belov. On Computing Minimal Correction Subsets. In Proc. of IJCAI. IJCAI/AAAI, 2013. (Cité en pages 25, 27, 28, 36 et 38.)

[Meseguer 2003] Pedro Meseguer, Noureddine Bouhmala, Taoufik Bouzoubaa, Morten Irgens et Martí Sánchez. Current approaches for solving over-constrained problems. Constraints, vol. 8, no. 1, pages 9–39, 2003. (Cité en page 47.)

[Monniaux 2008] David Monniaux. The pitfalls of verifying floating-point computations. ACM Transactions on Programming Languages and Systems (TOPLAS), vol. 30, no. 3, page 12, 2008. (Cité en page 112.)

[Patterson 2002] David Patterson, Aaron Brown, Pete Broadwell, George Candea, Mike Chen, James Cutler, Patricia Enriquez, Armando Fox, Emre Kiciman, Matthew Merzbacheret al. Recovery-oriented computing (ROC) : Motivation, definition, techniques, and case studies. Rapport technique, Technical Report UCB//CSD-02-1175, UC Berkeley Computer Science, 2002. (Cité en page 12.)

[Paul 2013] Dr. Jody Paul. Testing and Debugging. In `http://www.jodypaul.com/SWE/TD/TestDebug.html`, 2013. (Cité en page 2.)

[Reiter 1987] Raymond Reiter. A theory of diagnosis from first principles. Artificial intelligence, vol. 32, no. 1, pages 57–95, 1987. (Cité en page 41.)

[Renieris 2003] Manos Renieris et Steven P. Reiss. Fault Localization With Nearest Neighbor Queries. In Proceedings of ASE, pages 30–39. IEEE Computer Society, 2003. (Cité en page 14.)

[Rosenblum 1997] David S. Rosenblum et Elaine J. Weyuker. Lessons Learned from a Regression Testing Case Study. Empirical Software Engineering, vol. 2, no. 2, pages 188–191, 1997. (Cité en page 86.)

[Shapley 1952] Lloyd S Shapley. A value for n-person games. Rapport technique, DTIC Document, 1952. (Cité en page 113.)

[Sweeney 2015] Shahida Sweeney. Internet not designed for security, warns international expert. In `http://www.cio.com.au/article/569270/`

internet-designed-security-warns-international-expert/, 2015. (Non cité.)

[Tamiz 1996] Mehrdad Tamiz, Simon J. Mardle et Dylan F. Jones. Detecting iis in infeasible linear programmes using techniques from goal programming. Computers & OR, vol. 23, no. 2, pages 113–119, 1996. (Cité en page 29.)

[Wikipedia 2015] Wikipedia. List of software bugs — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=List_of_software_bugs&oldid=648559652, 2015. [Online ; accessed 3-March-2015]. (Non cité.)

[Wikipédia 2015] Wikipédia. Localisation automatique de bugs — Wikipédia, l'encyclopédie libre. http://fr.wikipedia.org/w/index.php?title=Localisation_automatique_de_bugs&oldid=116526583, 2015. [En ligne ; Page disponible le 25-juillet-2015]. (Non cité.)

[Wong 2009a] W Eric Wong et Vidroha Debroy. A survey of software fault localization. Department of Computer Science, University of Texas at Dallas, Tech. Rep. UTDCS-45, vol. 9, 2009. (Cité en pages 2 et 15.)

[Wong 2009b] W Eric Wong et Yu Qi. BP neural network-based effective fault localization. International Journal of Software Engineering and Knowledge Engineering, vol. 19, no. 04, pages 573–597, 2009. (Cité en page 114.)

[Wong 2010] W Eric Wong et Vidroha Debroy. Software Fault Localization. Encyclopedia of Software Engineering, vol. 1, pages 1147–1156, 2010. (Cité en pages 2 et 111.)

[Xuan 2014] Jifeng Xuan et Martin Monperrus. Test case purification for improving fault localization. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pages 52–63. ACM, 2014. (Cité en pages 10 et 15.)

[Zeller 2002] Andreas Zeller et Ralf Hildebrandt. Simplifying and isolating failure-inducing input. Software Engineering, IEEE Transactions on, vol. 28, no. 2, pages 183–200, 2002. (Cité en page 13.)

[Zhang 2006] Xiangyu Zhang, Neelam Gupta et Rajiv Gupta. Locating faults through automated predicate switching. In Proceedings of ICSE, pages 272–281. ACM, 2006. (Cité en page 18.)

Combinaison des techniques de Bounded Model Checking et de Programmation par Contraintes pour l'aide à la localisation d'erreurs

Résumé :
Un vérificateur de modèle peut produire une trace de contre-exemple, pour un programme erroné, qui est souvent difficile à exploiter pour localiser les erreurs dans le code source. Dans ma thèse, nous avons proposé un algorithme de localisation d'erreurs à partir de contre-exemples, nommé LocFaults, combinant les approches de Bounded Model-Checking (BMC) avec un problème de satisfaction de contraintes (CSP). Cet algorithme analyse les chemins du CFG (Control Flow Graph) du programme erroné pour calculer les sous-ensembles d'instructions suspectes permettant de corriger le programme. En effet, nous générons un système de contraintes pour les chemins du graphe de flot de contrôle pour lesquels au plus k instructions conditionnelles peuvent être erronées. Ensuite, nous calculons les MCSs (Minimal Correction Sets) de taille limitée sur chacun de ces chemins. La suppression de l'un de ces ensembles de contraintes donne un sous-ensemble satisfiable maximal, en d'autres termes, un sous-ensemble maximal de contraintes satisfaisant la postcondition. Pour calculer les MCSs, nous étendons l'algorithme générique proposé par Liffiton et Sakallah dans le but de traiter des programmes avec des instructions numériques plus efficacement. Cette approche a été évaluée expérimentalement sur des programmes académiques et réalistes.

Mots clés : Localisation d'erreurs, Programmation par contraintes, LocFaults, BugAssist, Minimal Correction Subsets, Minimal Correction Deviations.

Combining techniques of Bounded Model Checking and Constraint Programming to aid for error localization

Abstract :
A model checker can produce a trace of counter-example for erroneous program, which is often difficult to exploit to locate errors in source code. In my thesis, we proposed an error localization algorithm from counter-examples, named LocFaults, combining approaches of Bounded Model-Checking (BMC) with constraint satisfaction problem (CSP). This algorithm analyzes the paths of CFG (Control Flow Graph) of the erroneous program to calculate the subsets of suspicious instructions to correct the program. Indeed, we generate a system of constraints for paths of control flow graph for which at most k conditional statements can be wrong. Then we calculate the MCSs (Minimal Correction Sets) of limited size on each of these paths. Removal of one of these sets of constraints gives a maximal satisfiable subset, in other words, a maximal subset of constraints satisfying the postcondition. To calculate the MCSs, we extend the generic algorithm proposed by Liffiton and Sakallah in order to deal with programs with numerical instructions more efficiently. This approach has been experimentally evaluated on a set of academic and realistic programs.

Keywords : Error localization, Constraint Programming, LocFaults, BugAssist, Minimal Correction Subsets, Minimal Correction Deviations.